$2\sigma^6$

SEMANTIC DATABASE MODEL LANGUAGE (SDML):
GRAMMAR SPECIFICATION AND PARSER

by

RICHARD VERNON LANE

B. S., Michigan State University, 1980

A MASTERS THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

Approved by:

_____
Major Professor

CONTENTS

LIST OF FIGURES

Acknowledgements

I would like to give special thanks to my major professor, Dr. Beth Unger, for her support throughout this project. Also, I would like to thank my committee members, Dr. Bill Hankley and Dr. Auston Melton for their time. Additionally, thanks to Dr. David Schmidt for his assistance and knowledge of grammars and parser theory.

Finally, I would like to thank my mother and father for supporting me during my undergraduate study, thus enabling me to pursue further education.

Chapter 1: Introduction

2.1 Background

A database is a collection of elements (data items) and relationships between these elements which model some application environment. At any given point in time, the database will represent a state of the application environment. An update to the database represents a transition action taking the database from one state to another. A database model (or schema) provides a mechanism for representing the elements of a database along with their relationships. The structure of a database system should parallel the structure of the real world system it represents. This will allow for better user understanding of the database being designed, and allow the use of simple user front-ends, which adheres to the users understanding of the system.

The *semantic integrity* of a database is defined as how well the database adheres to the rules of the application environment. Hammer and McLeod [3] observe the following:

> "Every ... application environment has a set of rules which define its legitimate configurations. Any correct version of a data base must satisfy these rules .... The semantic integrity of a data base is violated when it ceases to represent a valid state of its application domain, because it fails to adhere to some of these rules."

Conventional database models (e.g., Network, Hierarchical, CODASYL, and Relational) have provided different methods of representing data and relationships in a database. Each of these database models, however, have been deficient in the semantic integrity of the data and relationships each represent. Conventional

database models have not adequately captured the meaning of data in a real world system. Hammer and McLeod state the following:

> "Conventional data models are not satisfactory for modeling data base application systems. The features that they provide are too low level and representational to allow the semantics of a data base to be directly expressed in the schema."

Some work has been done, however, to extend the relational database model [3][4][10] to capture more data meaning.

Semantic data models have been introduced to allow a database model to better describe the meaning of the data it represents. Semantic data models typically model the database application using some type of Data Definition Language (DDL). The DDL provided by a semantic data model allows the use of additional constructs to specify data meaning explicitly. McLeod and King [5] give four advantages of a semantic data model over conventional database models:

1. it allows a user-oriented formal specification and documentation aid to be established (in the form of a semantic schema),

2. it provides a basis for powerful, high-level user interface facilities,

3. it can serve as a conceptual database model in the database design and evolution process, more directly capturing the meaning of data then conventional database models, and

4. it can be used as the database model for a new kind of database management system, with increased functional capability and improved user interface characteristics (compared with conventional DBMSs).

Two such semantic data models are the Semantic Association Model (SAM) introduced by Su and Lo [7] and the Semantic Database Model (SDM) introduced by Hammer and McLeod [1]. SAM and SDM are similar in nature. Each uses a DDL to identify the entities in the database and list the attributes of each entity. The major difference is the method used to define the relationships between the entities in the database.

The Semantic Database Model (SDM) was introduced by Hammer and McLeod [1] as a higher-level database model which allows for greater semantic integrity in a database schema. SDM utilizes a formal specification approach to database modeling by providing a Data Definition Language (DDL). The SDM DDL, however, does not enforce a strict syntax. The SDM was previously only used as a documentation tool for database modeling and had no use for a strict grammar. Chapter 2 gives a detailed description of the SDM.

2.2 High Level Project Description

It can be seen from the DDL provided by the SDM that, with a few additional constraints, one could build a static data dictionary from the SDM specification. To perform this function automatically, the SDM specification would have to take on programming language qualities. The DDL provided by this specification must enforce a strict syntax to allow the DDL specification to be parsed by a data dictionary generation tool. Thus, the specification of a SDM Language (SDML) is desired.

The project involves the generation of a static data dictionary from an SDM specification. The database designer will be provided with a screen interface which will query for the information required to build an SDM for the database. The SDM specification will then be checked for syntactic and semantic correctness. Finally, the SDM specification will be used to build a static data dictionary for the database.

The primary use of this system will be in developing under-graduate skills in database development. The users of this tool will have a general knowledge of the SDM. The tool developed must be IBM PC compatible since the under-graduate

students utilizing the tool will be using an IBM PC compatible system.

The project logically partitions itself into three components:

1. User Interface.

   The user interface will provide the interactive interface into the system. The user of the tool will be queried for each class definition for the SDM. The user interface should be as user-friendly as possible while generating a SDM specification which is as syntactically sound as possible. This component should provide a listing of the SDM by class and a listing of the final SDM specification.

2. SDM Syntax Specification and Parser.

   The SDM described by Hammer and McLeod does not provide a strict syntax for its DDL. Since the SDM specification will, in this project, be used as an intermediate in the generation of a static data dictionary, a strict syntax is required. The specification of a BNF for the SDM specification is therefore required. Once this SDML is specified, a parser is required to verify the SDM generated by the user interface. The parser will verify correct syntax of the SDM generated as well as some semantic checks.

3. Data Dictionary Generation.

   Once a syntactically correct SDM is generated, the data dictionary must be generated. The data dictionary generation component will output a valid data dictionary of the database ordered by class.

2.3 Low Level Project Description

This paper will deal with the SDM syntax specification and parser. This will involve the development of a grammar for the SDM and the specification of the grammar in a Backus-Naur Form (BNF). This grammar will be referred to as the SDM Language (SDML). The basic grammar rules will stem from the SDM description as given in reference [1]. The grammar developed for SDML will be a deterministic context-free grammar. A deterministic grammar is one in which, given the next input token and the current state, only one rule application is possible. Thus, a deterministic grammar will support the design of a parser which requires a look ahead of only one input token. Chapter 4 will describe the design of the grammar for SDML.

Some extensions to the SDM will be included in the grammar for SDML. These extensions will be introduced and described in Chapter 3. These extensions will both enhance the existing SDM specification as well as provide some additional capabilities not provided by the SDM described by Hammer and McLeod. The additional capabilities introduced in this paper are intended to add to the semantic integrity of the SDM.

After the definition of SDML, the design of an SDML parser will be discussed in Chapter 5. The parser designed will be LR(1). This means that the input token stream will be processed left-to-right with a look ahead of one input token. This is the simplist form of parser to develop.

Chapter 2: SDM Description

This chapter will describe, in detail, the SDM as presented by Hammer and McLeod in reference [1]. Extensions to the SDM specification presented here will be introduced in Chapter 3. The SDML grammar definition will be discussed in Chapter 4.

An SDM specification is an organized grouping of entities which exist in an application's database. The structure of the SDM specification is as follows:

- Entities are logically grouped into *classes*. Each class in the SDM is, therefore, a logical collection of entities describing the class.

- Classes are related by *interclass connections*.

- Entities and classes have *attributes*. The attributes of an entity or class are what actually provide the semantic integrity for the SDM.

Consider the relational database model where the database is represented by a number of tables. Each table consists of a number of rows (tuples) and columns. The SDM class is synonymous with the relational database table. The entities in the SDM class represent the rows (tuples) in the relational database table. The attributes of the entities in the class represent the relational database table columns. Appendix 1 contains an example of an SDM schema. This schema represents car dealerships and cars in stock in each dealership. This schema will be used to support the discussion of some of the features of the SDM schema throughout the rest of this chapter.

- 7 -

3.1 Classes

A *class* is a logical collection of entities. Each class in SDM can be either a *base class* or a *nonbase class*. A base class is one which is defined independently of all other classes in the database. That is, it is not related in any way to any other class in the SDM schema. Therefore, base classes do not have interclass connections. The class CARS in Appendix 1 is an example of a base class. The class CARS cannot be described in terms of any other class in the SDM schema. A nonbase class is one which is described in terms of one or more other classes in the SDM schema. This relationship is defined with the *interclass connection*. For example, the nonbase class BUICKS is defined as a subclass of the base class CARS. The entities in any nonbase class, therefore, are dependent on the entities in the classes in which they are based.

Classes in the SDM schema can represent any one of the following abstractions:

   i. Concrete object such as CARS or DEALERS.

  ii. Events such as car preparations (PREPS).

  iii. Categorizations or aggregations of entities.

  iv. Names (syntactic identifiers).

The first three types of classes represent definitions of base and nonbase classes and will be described in the following sections. The last type of class (name classes) represent the data in the SDM schema which is communicated with the outside world (i.e., the database user). The name class defines the domain of the data item which will be used to enter data into and retrieve data from the database. The

- 8 -

first three types of classes will define the value class of their attributes as one of the name classes defined elsewhere in the SDM schema. Name classes will be defined in detail in section 2.4.

Each class in the SDM schema has the following features:

1. A *class name*, which uniquely identifies the class in the SDM schema. Multiple synonymous class names are permitted. Class names in this paper are represented by a string of upper-case letters and special characters (e.g., CARS and BUICKS).

2. A collection of *members*, which represent the entities in the class. The concept of class members is implicit in the definition of the SDM schema. That is, class members are not explicitly defined in the SDM schema.

3. An optional *class description*, which gives an English language description of the class. The class description allows for better documentation of the SDM schema.

4. A collections of *attributes*, which describe the members of the class or the class as a whole. The two types of attributes are:

   a. *Member attributes* describe a member of a class by logically associating the class member to one or more entities in the same or another class.

   b. *Class attributes* describe the given class as a whole. Attributes are described in section 2.3.

5. A base class has a list of *identifiers* which uniquely identify a member of a class. Thus, the identifiers serve as a "key" for uniquely identifying a specific

- 9 -

member (or entity) of a class. Use of identifiers implies that duplicate members are not allowed (see item 6).

6. A base class is specified as either *containing duplicates* or *not containing duplicates*. This defines whether the class can contain duplicate members or not. The default is that duplicate members are allowed.

7. A nonbase class is defined to be related to one or more other classes by specification of an *interclass connection*. The interclass connection is described in the following section.

3.2 Interclass Connection

Any nonbase class has an interclass connection describing how the nonbase class relates to other classes in the database. Interclass connections define the entities in a nonbase class by specifying a predicate P which, when applied to the entities of another class C, yield the desired nonbase subclass S. The two types of interclass connections are the *subclass connection* and the *grouping connection*.

3.2.1 *Subclass Connection*. This type of interclass connection is used to define a nonbase class S as a subclass of a parent class C. Thus, the members of the nonbase class S form a subset of the members of the parent class C. The members of the subclass S are determined by the definition of a predicate P which is applied to the members of C. The subclass connection defines the nonbase subclass S as consisting of all entities of the parent class C which satisfy a given predicate P. The following sections describe the four types of subclasses which can be defined via subclass connections.

3.2.1.1 *Attribute-Defined Subclass.* This type of subclass connection defines the subclass S to be all members of C such that the given attribute(s), A_i of C satisfy a given regular expression involving a value V_{A_i}, of the value class of A_i. An *attribute predicate* is used to define this type of subclass connection. The attribute predicate can be either a *simple predicate:*

where $<A_1> = <V_{A_1}>$,

or a *compound predicate:*

where $<A_1> = <V_{A_1}>$ and $<A_2> = <V_{A_2}>$.

Any of the logical operators and scalar comparators can be used in the attribute predicate where the attribute is single valued. The attribute predicate can also be a set operation:

where $<A_1>$ contains $<V_{A_1}>$.

where the attribute is multivalued.

An example of the use of this type of subclass connection is in the definition of the class BUICKS which is a subclass of CARS. The interclass connection for the BUICKS nonbase class would be

subclass of *CARS* where *Make* = *'buick'* .

Here, the BUICKS nonbase class is made up of all members of class CARS where the "Make" attribute has a value of 'buick'.

3.2.1.2 *User-Controllable Subclass.* This type of subclass connection defines the subclass S to be any members of the class C as specified by the database administrator. The predicate

where specified

is used to define this type of subclass connection.

An example of the use of this type of subclass connection is in the definition of the nonbase class PREPARED_CARS which is a subclass of CARS. The interclass connection for the PREPARED_CARS nonbase class would be

subclass of *CARS* where specified.

Here, the members of the PREPARED_CARS class would be CARS which are prepared for delivery (as specified by the database administrator).

3.2.1.3 *Set-Operator-Defined Subclass*. This type of subclass connection defines the subclass S to be any members of C which adhere to some set operation on two other database classes C_1 and C_2. Possible set operations are *intersection*, *union*, and *difference*. The classes C_1 and C_2 must be subclasses of class C for the set operations to be valid. The predicates

is in $<C_1>$

and

is not in $<C_1>$

are used to define a set-operator-defined subclass.

An example of the use of this type of subclass connection is in the definition of the nonbase class PREPARED_BUICKS, which is a subclass of CARS. The interclass connection for the PREPARED_BUICKS nonbase class would be

subclass of *CARS* where is in *PREPARED_CARS* and is in *BUICKS*.

This interclass connection also describes the use of the intersection set-operator-

defined subclass.

3.2.1.4 *Existence Subclass*. This type of subclass connection defines the subclass S to be any members of C which are currently values of some attribute A of another class C_1. The predicate

where is a value of A of C_1

is used to specify a nonbase class S as an existence subclass of C.

An example of the use of this type of subclass connection is in the definition of the nonbase class BUICK_DEALERS, which is a subclass of DEALERS. The interclass connection for the BUICK_DEALERS nonbase class would be

subclass of *DEALERS* where is a value of
Dealership of *BUICKS* .

3.2.2 *Grouping Connection*. The grouping connection defines a nonbase class which has members which are classes. The nonbase grouping class G defines a class which has members which are classes of the underlying class U. Thus, grouping classes are of a higher order than subclasses. In the relational database model, they would be representational of a table of tables. The following sections describe the three types of grouping classes which can be defined via the grouping connection.

3.2.2.1 *Expression-Defined Grouping Class*. This type of grouping class allows definition of a nonbase class G which is made up of all classes formed by collecting the members of the underlying class U into classes based on a common value for one or more member attributes of U. The predicate

on common value of $<attribute>$

is used to specify this type of grouping class.

- 13 -

Note that each class in the grouping class is also an attribute-defined subclass of the underlying class U. If this attribute-defined subclass is explicitly defined in the SDM schema elsewhere, the qualifier

groups defined as classes are $C_1, C_2, ..., C_n$

is used to state that this explicit nonbase class definition is already made. This implies a duplicate class definition, one explicitly made in the SDM schema, and one as a class member of a grouping class.

An example of the use of this grouping class is in the definition of the nonbase class CAR_MODELS. This class defines a grouping of all of the different car models. The predicate used to define CAR_MODELS is

grouping of *CARS* on common value of *Make* and *Model*.

Since the attribute-defined subclass SOMERSETS is defined explicitly in the SDM schema, the qualifier

groups defined as classes are *SOMERSETS*

is also stated.

3.2.2.2 *Enumerated Grouping Class.* This type of grouping class allows definition of a nonbase class G which is made up of all the given classes. Thus, G is defined as a grouping of classes $C_1, C_2, ..., C_n$, where each of the classes, C_i, are subclasses of the underlying class U. The predicate

consisting of classes $C_1, C_2, ..., C_n$

is used to define this type of grouping class.

3.2.2.3 *User-Controllable Grouping Class.* This type of grouping class allows definition of a nonbase class G which is made up of a user defined number of classes, each with a user defined number of members. The predicate

>   where specified

is used to define this type of grouping class.

3.3 Attributes

Each class in the SDM schema has a collection of attributes (representing the columns in the relational database model) describing the class. Attributes can either describe the members of a class (member attributes) or the class itself (class attributes). A member attribute has a value for each member (or each tuple) of the class. A class attribute has only one value for the class as a whole (independent of the number of members of the class).

Each attribute has the following features:

1. An *attribute name* which uniquely identifies the attribute within the class, the underlying base class, and all eventual subclasses of the underlying base class. As with class names, multiple synonymous names are permitted.

2. A *value* which is either an entity in the database or a collection of entities. An attributes *value class* defines the class which the value comes from. An attributes value can also be *null*.

3. An optional *attribute description* is an English language description of the attribute. This serves as a documentation tool in the SDM schema.

4. The attribute can be either *single valued* or *multivalued*. The default is single valued. A single valued attribute defines the value of the attribute as a single tuple of the value class. A multivalued attribute defines the value of the attribute as a collection of tuples of the value class. Therefore, the multivalued attribute is itself an SDM class.

   A multivalued attribute can also have a constraint on the size of the class (i.e., number of members) placed by specifying multivalued with size between X and Y", where X and Y are integers.

5. An attribute value can be specified as *mandatory*, which means that it cannot contain a null value. The term may not be null is used to specify a mandatory attribute value.

6. An attribute can be specified as *not changeable*. Here, once a non-null value is specified for an attribute, it cannot be changed.

7. An attribute can be specified to be *exhaustive* of its value class. Here, every value of the value class must be the attribute value of some entity in the class. The phrase exhausts value class is used to specify this feature.

8. A multivalued attribute can be specified as having *nonoverlapping* values. This means that no two values of the attribute have any entities (tuples) in common. The term no overlap in values is used to specify nonoverlapping values.

9. An attribute can be related to other attributes in the SDM schema with *attribute relationships*. These attribute relationships are described in the following sections.

3.3.1 *Attribute Mappings.* Attribute *mappings* provide a mechanism of directly referencing an attribute within the value class of an attribute. An attribute mapping is specified by listing each attribute in the sequence separated by periods. An example of an attribute mapping is when referencing the name of a dealership for cars. Here, the attribute mapping "Dealership.Name" references the "Name" attribute for the DEALERS class, which is the value class for the "Dealerships" attribute of the CARS class. Thus, "Dealership.Name" gives the dealership name for the given car.

Formally, an attribute mapping is a sequence of attributes, N_i defined within classes C_i respectively $(N_1.N_2. \cdots N_m)$ where the value class of N_i is C_{i+1} for all $i = 1,...,m-1$.

The following rules apply to attribute mappings:

i. An attribute mapping *AM* is *multivalued* if any one of the attributes N_i is multivalued.

ii. the value class of the attribute mapping *AM* is the value class of the last attribute N_m in the mapping (i.e., class C_m).

3.3.2 *Attribute Inheritance.* When defining a subclass S of a class C, the attributes of class C are inherited by the subclass S according to the following rules:

1. A class S defined as an attribute-defined subclass or a user-controllable subclass of a parent class C inherits all of the member attributes of C.

2. A class S defined as an intersection subclass of classes C_1 and C_2 inherits all

of the member attributes of both classes C_1 and C_2.

3. A class S defined as a union subclass of classes C_1 and C_2 inherits all of the member attributes common to C_1 and C_2.

4. A class S defined as a difference subclass of classes C and C_1 (i.e., members of class C not in class C_1) inherits all of the member attributes of class C.

Member attributes inherited by a subclass are not explicitly defined, but are assumed based on how the subclass is defined (by the above rules). One can, however, place constraints on the value class of an inherited member attribute by explicitly specifying the inherited member attribute in the subclass definition with the new value class specification. An example of this is in the specification of the subclass BUICKS of class CARS. Here, an additional constraint is put on the inherited member attribute "Model" of the BUICK subclass. The constraint is that the "Model" attribute of class BUICKS can only be from the value class BUICK_MODELS (instead of the inherited value class CAR_MODELS).

3.3.3 *Member Attribute Interrelationships.* Member attribute interrelationships allow member attributes of two or more classes to be related in the SDM schema. The three types of member attribute interrelationships are the *inverse*, the *match*, and the *derivation*. The inverse and matching interrelationships are described here. The derivation is described in the next section. Formal definitions are also given for the inverse and matching interrelationships, since they may be easier to comprehend then the english definition.

The first member attribute interrelationship is the *inverse* relationship. The inverse relationship defines a symmetrical relationship between two attributes of

- 18 -

two classes. Consider classes C_1 and C_2, with attributes A_1 and A_2 respectively. Member attribute A_1 of C_1 is defined as the inverse of attribute A_2 of C_2 if the value of A_1 for a member M_1 of C_1 consists of those members of C_2 whose value of A_2 is M_1. Since inversion establishes a symmetrical relationship, attribute A_2 of C_2 is also defined as the inverse of attribute A_1 or C_2 in the SDM schema.

Formally, the value of A_1 for member M_1 of class C_1 is defined in terms of attribute A_2 of class C_2 as follows:

A_1 of $M_1 \equiv$ members of C_2 such that $M_1 \in A_2$

The SDM schema segment in Figure 3-1 describes how the inverse relationship would be described in an SDM schema. The value class restrictions are also displayed.

```
C₁
   member attributes:
       A₁
          value class: C₂
          inverse: A₂
          .
          .
          .
C₂
   member attributes:
       A₂
          value class: C₁
          inverse: A₁
```
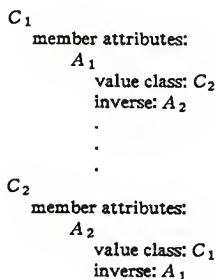
Figure 3-1. SDM Inverse Attribute Relationship

Operationally, the inverse works as follows. When class C_1 (or C_2) is changed in some way (i.e., member record added, deleted, updated, etc.), the corresponding appropriate will be made to members of class C_2 (or C_1). Thus, the inverse relationship ensures that the two attributes remain consistently defined throughout the life of the database.

An example of the inverse relationship is that of the attribute "Dealership" of class CARS and attribute "Cars_in_stock" of class DEALERS. Here, the value of the "Dealership" attribute of a car is those DEALERS whose "Cars_in_stock" contain the given car. Also, the value of the "Cars_in_stock" for a dealer contain those members of CARS which have a "Dealership" attribute value of the given dealership. The inverse relationship would ensure that the "Cars_in_stock" attribute of DEALERS is updated automatically when members are added to class CARS which specify the appropriate value of the "Dealership" attribute.

The second attribute interrelationship is the *matching* relationship. The value of the match attribute A_1 for the member M_1 of class C_1 is determined as follows:

a. A member M_2 of some class C_2 is determined such that C_2 has M_1 as its value of member attribute A_2.

b. The value of member attribute A for M_2 is used as the value of A_1 for M_1.

The matching relationship is specified in the SDM schema as shown in Figure 3-2.

```
C
    member attributes:
        A₁
            value class: C₁
            match: A of C₂ on A₂
            .
            .
            .
C₂
    member attributes:
        A₂
            value class: C
        A
            value class: C₁
```
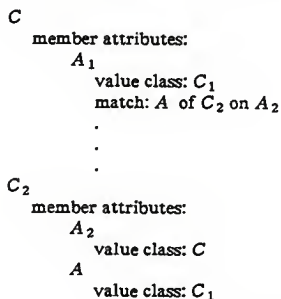
Figure 3-2. SDM Attribute Matching Relationship

Formally, the value of the A_1 attribute for member M_1 of class C_1 is determined

from attributes A and A_2, and member M_2 of class C_2 as follows:

A_1 of $M_1 \equiv A$ of M_2 in C_2 such that $M_1 \in A_2$ in M_2

The matching attribute, therefore, automatically derives the value of attribute A_1 of class C_1 from the given information. This implies that the attribute A_1 can never be given a value directly by the user.

An example of the use of the matching relationship is in the definition of the "Owner" attribute of the CARS class. Here, the owner of a car is the same as the owner of the dealership where the car is in stock. To specify this relationship, the "Owner" attribute of a car is defined to be the "Owned_by" attribute value for the member of DEALERS which has this car as a value of "Cars_in_stock". Notice that the class of "Owners" of class CARS and "Owned_by" of class DEALERS must be the same.

3.3.3.1 *Member Attribute Derivations.* The third and last type of member attribute interrelationship is the *derivation*. The derivation is used to define an attribute as being derived (or calculated) from other attributes in the SDM schema. A number of derivation primitives are supported by the SDM. Each derivation primitive supplies a mechanism for computing a derived attribute. Combined use of these derivation primitives can lead to the development of arbitrarily complex derivations.

A description of all the derivation primitives supported by SDM follows. The descriptions involve the definition of a derived attribute A_1 of class C_1 with member M_1.

1. A_1 can be defined as an *ordering* attribute. Here, the value of attribute A_1 is

the sequential position of M_1 within C_1 when the members of C_1 are ordered by other specified attributes of C_1. Members of C_1 can be ordered by increasing (default) or decreasing value. The phrase

order by A_2

is used to define an ordering derivation. Ordering of the members M_i of class C_1 can also be specified in groups based on another attribute A_3 of C_1. Here, the value of A_1 is the sequential position of member M_1 within the group of members of C_1 which have a common value of attribute A_3. The phrase

order by A_2 within A_3

is used to specify ordering withing groups.

2. A_1 can be defined as an *existence* attribute. Here, A_1 contains the value "yes" if member M_1 of C_1 is a member of some other specified class C_2, and "no" otherwise. The phrase

if in C_2

is used to define an existence attribute.

3. A_1 can be defined by recursively tracing the values of some attribute A_2. The value class of A_2 must be the same as the value class of A_1. The phrase

all levels of values of A_2

is used to define this type of attribute derivation. This attribute derivation can also specify a limit to the number of recursions of tracing the values of A_2. The phrase

up to n levels of values of A_2

is used to place a restriction of n levels on the recursive tracing of attribute A_2.

4. The derived multivalued member attribute *Contents* is automatically defined when a grouping class is defined. This member attribute has a value representing the contents of each class underlying the grouping class. That is, each value of "Contents" represents all of the members of one of the classes underlying the grouping class.

5. A_1 can be defined to be directly derived from another attribute A_2. Here, whatever value is given to attribute A_2 is also given to attribute A_1. The description

same as A_2

is used to define this type of derivation.

6. A_1 can be defined as a *subvalue* of some other attribute A_2 which satisfies some predicate P. The predicate, P, can be any of the attribute predicates defined in section 2.2.1. The description

subvalue of A_2 where P

is used to define a subvalue attribute. Predicate P may contain mappings which are used to determine which values of A_2 are applicable. These mappings, however, must be consistent with the value class of attribute A_2.

7. A_1 can be defined as the intersection, union, or difference of two other multivalued attributes. A union derivation would be specified as

where is in A_2 or is in A_3.

8. A_1 can be defined in terms of other attributes with an arithmetic expression. All of the attributes involved in the arithmetic expression must have a value class of the built in class NUMBERS. The set of possible arithmetic operators are addition ("+"), subtraction ("-"), multiplication ("*"), division ("/"), and exponentiation ("!").

9. A_1 can be defined to be the "minimum", "maximum", "average", or "sum" of another multivalued attribute. The defining attribute must have a value class of the built in class NUMBERS.

10. A_1 can be defined to be the number of members in some other multivalued attribute A_2. The phrase

> number of members in A_2

is used to specify this derivation. The user can also specify that A_1 be the number of unique members of attribute A_2. The number of unique members differs from the number of members only when duplicates are present in the attribute A_2.

3.3.3.2 *Member Attribute Definition Rules.* Hammer and McLeod describe how the use of derivation specifications must be used to avoid inconsistent attribute specifications. Every attribute, A_1, satisfies one of the following cases:

a. A_1 has exactly one derivation. Here, the value of A_1 is completely specified by the derivation. If an inverse of A_1 exists, it may not have a derivation or a matching specification.

- 24 -

b. A_1 has exactly one matching specification. Here, the value of A_1 is completely specified by its relationships with an entity (or entities) to which it is matched. If an inverse of A_1 exists, it may not have a derivation. The inverse of A_1 can, however, have a matching specification, but must be consistent with the matching specification of A_1.

c. A_1 has neither a matching specification nor a derivation. Here, it may be the case that the inverse of A_1 has a matching specification or a derivation. If this is the case, one of the above two cases applies. Otherwise, A_1 and A_2 form a pair of primitive values that are defined in terms of one another, but which are independent of all other information in the database.

The concept of defining an inverse and a matching relationship within the same attribute definition warrants some discussion. Recall from the discussions of the inverse and matching attribute interrelationships, the operation of the two types of relationships. Inverse supplies an automatic update mechanism for one attribute when the other corresponding attribute is modified. Matching supplies an automatic derivation of an attribute from other information in the database. Consider an attribute A_1 defined in class C_1 with both an inverse and matching attribute interrelationship defined as shown in Figure 3-3.

C_1
  A_1
      value class: C_2
      inverse: A_3
      match: A of C_3 on A_2
  .
  .
  .
C_2
  A_3
      value class: C_1
      inverse: A_1
  .
  .
  .
C_3
  A
      value class: C_2
  A_2
      value class: C_1
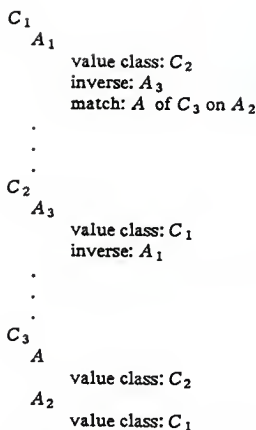
Figure 3-3. Combined Use of Match and Inverse Attribute Interrelationships

The inverse relationship between A_1 and A_3 means that whenever one of the two attributes is changed, the other one is changed appropriately. The match relationship for A_1 means that A_1 cannot ever be directly assigned a value, but is automatically derived from attributes A and A_2 in class C_3. This implies an indirect dependence of A_3 of class C_2 on attributes A and A_2 in class C_3. If this dependence were not recognized by the model, a change to attribute A_3 for class C_2 would initiate a corresponding change to attribute A_1 for class C_1. This could invalidate the matching relationship between attribute A_1 and the attributes in class C_3. Therefore, when a matching relationship is defined within an attribute definition, and an inverse relationship also exists, a matching relationship is also assumed for the inverse attribute (here A_3). Further, if the database designer specifies a matching relationship for the inverse attribute, it had better be the matching relationship which is assumed.

The matching relationship which is assumed for A_3 is

match: A_2 of C_3 on A

To see this (and in fact to conceptualize the inverse/match combined definition), an example will be given. Referring to Figure 3-3, consider the following notation:

i. $M_i^{C_j}$ will represent member i of class C_j.

ii. $M_i^{C_j}(A_k:M_n^{C_p}, \cdots)$ means that the kth attribute of member $M_i^{C_j}$ contains the members M_n, etc.. from class C_p. Note that this implies that the value class of A_k is C_p.

Figure 3-4 shows a derivation for the attributes A_1 and A_3 based on the values of A and A_2 in class C_2. First, given the values for A and A_2 for members $M_1^{C_3}$ and $M_3^{C_3}$ of class C_3, and using the formal definition for matching, we can attempt to derive the A_1 values for all members of class C_1. Given only the two specified members of class C_3, only the A_1 attribute for member $M_1^{C_1}$ is derivable since this is the only member of C_1 which appears as a member of the A attribute of C_3. Using the formal definition of matching, the A_1 attribute for member $M_1^{C_1}$ is the two members $M_2^{C_2}$ and $M_3^{C_2}$ of class C_2 (which is the value class of A_1). Now applying the inverse relationship between A_1 and A_3, we can attempt to derive the A_3 attribute value for all members of class C_2. Using the formal definition for inverse, members $M_2^{C_2}$ and $M_3^{C_2}$ both have $M_1^{C_1}$ as their value (note that C_1 is the value class of A_3, which is consistent with the definition). Now, it can be seen that using the assumed matching definition for attribute A_3, the values derived for A_3 using the inverse with A_1 are exactly the same as the matching relationship between A_3 and attributes A and A_2 of class C_3.

$C_1:$

$\quad M_1^{C_1} (A_1 : M_2^{C_2}, M_3^{C_2})$
$\quad M_2^{C_1} (A_1 : ?)$

$C_2:$

$\quad M_1^{C_2} (A_3 : ?)$
$\quad M_2^{C_2} (A_3 : M_1^{C_1})$
$\quad M_3^{C_2} (A_3 : M_1^{C_1})$
$\quad M_4^{C_2} (A_3 : ?)$

$C_3:$

$\quad M_1^{C_3} (A : M_2^{C_2}, A_2 : M_1^{C_1})$
$\quad M_2^{C_3} (A : ?, A_2 : ?)$
$\quad M_3^{C_3} (A : M_3^{C_2}, A_2 : M_1^{C_1})$

Figure 3-4. Example of Match/Inverse Combined Derivation

This example is by no means a proof of the assumed matching relationship, but only shows how the assumed matching relationship logically fits in with the inverse and matching relationships explicitly defined.

3.3.4 *Class Attribute Derivations.* Attribute derivation primitives analogous to those listed in items 5 through 10 in the previous section can be used to define class attribute derivations. Instead of using other member attributes to derive the member attribute, other class attributes are used to derive the class attribute. In addition to items 5 through 10 of the previous section, two additional primitives can be used to derive a class attribute:

1. The class attribute can be defined to be the number of members in the class in which it resides. The phrase

    number of members in this class

    is used to define this type of class attribute derivation.

2. The class attribute can be defined to be the "minimum", "maximum", "average", or "sum" of one of the member attributes in the class in which it

resides. The phrase

sum of A over members of this class

is an example of this type of class attribute derivation.

3.4 Name Classes

Name classes define the *domain* of the data which is used to communicate with the outside world; that is, the data which can be entered by a user and which is supplied to the user as a response to queries, etc. A name class in SDM is a subclass of one of the built-in classes, formed by application of some predicate, P, to to the built-in class. The predicates which can be used to define a name class are as follows:

1. The name class can be defined as a subclass of a built-in class. Here, all legal members of the built-in class can be entered.

2. The name class can be defined as a subclass of a built-in class, where specified by the database administrator. Here, the members of the name class are those members of the built-in class which are specified by the database administrator.

3. The name class can be defined as a subclass of STRINGS which follows some specified format. Here, the name class is defined as all members of STRINGS which satisfy some data format directive which is specified in the SDM schema. This name class represents a class of STRINGS which satisfy the format directive; it will not actually have every member of STRING satisfying the format directive. The next section details the use of the format directive, F, when used in this type of name class definition.

3.4.1 *Format Directives* The format directive is defined in terms of the Domain Definition Language (DDL) as detailed in reference [13]. DDL was created to allow a relational database model to specify the semantics of the data which it represented. In the SDM schema, DDL is used to specify the format directive, F, used in the subclass interclass connection when defining a name class. The format directive (as in the domain definition of reference [13]), establishes the domain of the name class when it is defined in the SDM schema; that is, the domain of the name classes are static (do not change).

McLeod defined a domain definition to consist of four parts: (1) domain name, (2) domain description (establishing the domain), (3) ordering of the domain values, and (4) a violation-action to be taken if the domain description is violated. In the SDM schema, the domain name will be the class name of the name class. The domain description will be for format directive in the name class. Also, as part of the format directive, the ordering of the domain values defined in the domain description will be supplied.

A domain description is specified by any one (or a combination of) the following:

    a. decomposing the domain values into *subunits* which restrict the domain values for the name class by specifying a format for the data,

    b. enumerating the domain values; that is, specifying that the domain consists of a finite number of values (note that the enumeration of the domain values is done at the subunit level in the format directive),

    c. placing restrictions on the set of domain values by definition of a predicate, p, which limits the set of domain values.

The format directive can also be a number of alternative domain decompositions or enumerations for the domain of the name class. Each subunit specification can be prefixed with a *label* which allows reference to the value of the subunit within the predicate p.

The format directive can specify that the subunits defined be ordered by some criteria. The possible ordering alternatives are as follows:

i. list of subunits which define the ordering precedence for the domain value when compared,

ii. no ordering ("none") which implies that only equality comparators are possible,

iii. atomic ordering; that is, the when the domain value is compared, numeric ordering is used for real numbers and lexicographic ordering is used for character strings.

A domain violation action can be supplied to determine a course of action if data is entered by the user which does not satisfy the given format directive. The DDL presented in reference [13] provides three basic domain violation actions:

1. flag an error and supply an optional message,

2. substitute a known value for the value in error, or

3. call a user-defined procedure to handle the error.

The basic format of a format directive for an interclass connection of a name class is shown in Figure 3-5.

subclass of STRINGS where format is
        $[label_{1a}:]$ *subunit* $_{1a}$
        $[label_{2a}:]$ *subunit* $_{2a}$
        .
        .
        .
[or
        $[label_{1b}:]$ *subunit* $_{1b}$
        $[label_{2b}:]$ *subunit* $_{2b}$
        .
        .
        .]

[where p]
[ordering: ordering-spec]
[violation action: violation-action-spec]

Figure 3-5. SDM Format Directive Specification

An example of the use of the format directive in the definition of name classes is in the definition of the name class VIN_CODES in Appendix 1. The exact syntax of the format directive will be detailed in Chapter 4.

3.5 Built-In Classes

For convenience, SDM provides some built-in class definitions. The built-in classes provided by SDM are as follows:

YES/NO   This class has the two members "yes" and "no". This provides the boolean class definition.

REALS   This class has members representing all real numbers.

INTEGERS This class has members representing all integers (positive and negative).

NUMBERS This class has members representing all members of the REALS and INTEGERS classes.

STRINGS This class has members representing all possible strings of characters.

3.6 Summary

This chapter has detailed the SDM as introduced by Hammer and McLeod in reference [1]. Chapter 3 will introduce extensions to the SDM which will provide greater capabilities as well as provide more semantic integrity into the SDM described in this chapter. Chapter 4 will introduce the reader to formal language theory and present the SDML grammar. Finally, Chapter 5 will discuss parsing of the SDML grammar and include a discussion of the semantic checks which should be made in the SDML parser which would not be caught in the SDML grammar itself.

Chapter 3: SDM Extensions

This chapter describes the extensions to the SDM model which are included in the SDML grammar presented in Chapter 4. Since the SDM has been used in classroom study for some time, the SDML grammar was designed to be as syntactically close as possible to the SDM presented in Chapter 2. The two major changes to the SDM are:

1. Description text in SDML is enclosed in curly-brackets. This is necessary to be able to recognize the end of the description (since any character or reserved words may appear in the description).

2. The equality predicate (item 9 in section 2.3.3.1) and the set-order-derived predicate (item 10 in section 2.3.3.1) were combined to form an enhanced equality predicate. The equality predicate allowed an arithmetic expression involving attribute mappings but did not allow set operators on multivalued attribute mappings. The set-order-derived predicate allowed a single set operator to be applied to a multivalued attribute mapping. SDML will allow an arithmetic expression involving attribute mappings and set operators on multivalued attribute mappings. Note that a simple equality predicate involving a single set operator on a multivalued attribute is exactly the set-order-derived predicate. Thus the reason for the merging of the equality predicate and the set-order-derived predicate.

Appendix 2 contains the structure of an SDML specification in a informal syntactic format. This structure specification is provided to allow the reader to gain a general understanding of the structure of an SDML specification without

attempting to follow the BNF specification.

The DDL presented in reference [13] was also included into the SDML grammar with some modifications. Reference [13] presented a BNF notation for the DDL. The BNF listed, however, included some constructs which resulted in a non-LR(1) grammar. Therefore, some additional syntax was added to the DDL which made it LR(1). Changes to the DDL will be noted in the next Chapter as the BNF for the DDL within SDML is presented.

The following sections describe some extensions which, when added to the SDM described in Chapter 2, will add some additional capabilities. Appendix 3 contains the structure of the DDL included in SDML.

4.1 Assertions

Assertions will allow the SDM designer to include statements of fact about the database being modeled. An *assertion* is a statement of fact which should always remain true independent of the state of the database. If the database enters a state such that the assertion is no longer true, the data in the database is in error. An assertion *failure action* is supplied to indicate the action necessary when the corresponding assertion fails. These assertions are considered run-time assertions which will flag invalid database states during actual database use. Any number of assertions can be defined in the SDM schema.

Three types of assertions are supplied to the SDM designer:

- Member Attribute Assertions,

- Class Attribute Assertions, and

- Interclass Assertions.

With each of the three types of assertions comes a choice of three failure actions:

1. Flag the database state as an error state and optionally print a message about the condition.

2. Warn the database administrator about the condition with a message.

3. Call a user-defined procedure which will take the appropriate action. This alternative allows the database designer to build in notifications when certain data values in the database obtain specified values. The optional error message is not supplied on this option since the user-defined procedure should supply any failure messages necessary.

A failure action of *error* is assumed if no failure action is given for the assertion.

A third failure action could also be supplied which would allow the database designer to indicate how the database should be automatically repaired when the assertion fails. For example, a member attribute assertion failure action could be supplied such that the member attribute value is substituted with a derived MEMBER_ATTRIBUTE_DERIVATION (see Appendix 2). A class attribute assertion failure action could indicate that the class attribute value is substituted with a derived CLASS_ATTRIBUTE_DERIVATION (see Appendix 2). Finally, an interclass assertion failure action could be supplied such that the class is re-populated using some other interclass connection. More sophisticated automatic repair specifications could be employed to detect the source of the problem and correct any other attributes which are found in error. Automatic database repair specifications within an SDML specification will not be dealt with here since they

lend themselves to much more study then could be supplied in this paper. They are, however, a possible topic for further research.

The automatic repair of an invalid database state could also be thought of as a function of the user-defined procedure. This way, the procedure specified would be responsible for detecting the source of the error, correcting the error, and re-verifying the state of the corrected database.

4.1.1 *Member Attribute Assertions* Member attribute assertions assert something about the value of the given member attribute in terms of other member attributes, class attributes for that class, or constants. The member attribute assertion can either

1. specify an arithmetic expression which should always evaluate to "true" independent of the current database state, or

2. specify a procedure to call to provide the necessary assertion checking.

The second type of assertion allows the database designer to indicate that the attribute value must be checked in a non-trivial manner which can be described by a procedure interface.

An example of the use of the first type of member attribute assertion is asserting that the Date_of_preparation attribute of class SCHEDULED_PREPS is always $\leqslant$ CURRENT_DATE. The CURRENT_DATE used here is another extension of the SDM schema and is presented in section 4.3. This also demonstrates the use of the attribute assertion to notify the database administrator when the value of an attribute takes on some specified values or exceeds some specified range of values. This example illustrates the use of the so called notification-assertion by notifying

- 37 -

the database administrator when a car is past due for preparation. This database state is not actually in error, but requires some further action on the record to resolve the conflict (e.g., either to change the preparation date or to prepare the car for delivery and move the record to PREPARED_CARS class). The Date_of_preparation member attribute of class SCHEDULED_PREPS Here the assertion

    assertion: Preparation_date ≤ CURRENT_DATE
    failure action: call _notify_past_due

is given within the member attribute "Date_of_preparation" of class SCHEDULED_PREPS.

An example of the second type of member attribute assertion occurs when specifying an interior color for a car. Typically, only certain interior colors can be used based on the exterior color of the car. Since the selection of interior colors based on the exterior color is non-trivial, an assertion can be supplied to verify the choice of interior colors based on the exterior color. Here the assertion

    assertion: call _verify_interior_color
    failure action: error 'invalid exterior/interior combination'

is given within the member attribute "Interior_color" of class CARS to show this relationship between interior and exterior colors.

4.1.2 *Class Attribute Assertions* Class attribute assertions assert something about the value of the given class attribute in terms of other class attributes, member attributes within that class, or constants. As with member attribute assertions, the assertion can either specify an arithmetic expression which should evaluate to "true" independent of the current database state, or provide a procedure call which verifies the current database state.

An example of the use of the class attribute assertion is asserting that the Number_of_cars_sold attribute of class CARS_SOLD is exactly equal to the number of members in class SCHEDULED_PREPS plus the number of members in class PREPARED_CARS. Figure 4-1 shows how the class definition of CARS_SOLD would be defined to show this assertion. The *sizeof* class operator is another extension to the SDM schema and is presented in section 4.4.

4.1.3 *Interclass Assertions* Interclass assertions assert something about the interclass connection for a nonbase class. This is used to ensure that other interclass connections are also true for the nonbase class. Thus, the interclass assertion specifies one or more other interclass connections which must also be true. The interclass assertion would be most beneficial when dealing with classes which are user-controllable classes; that is, the database administrator populates the class. An example of the use of the interclass assertion to verify user-controllable classes is shown in Figure 4-1.

CARS_SOLD
    description: { All cars sold to customers. }
    interclass connection: subclass of CARS where
                            specified
    interclass assertion: subclass of CARS where
                            is in PREPARED_CARS or
                            is in SCHEDULED_PREPS
    failure action: warning
                'car not scheduled for preparation'
    member attributes:

    Sold_to
        value class: PERSON_NAMES

    Sold_by
        value class: PERSON_NAMES
        assertion: Sold_by contained in
                    Dealership.Employees
        failure action:
            error 'Salesman not employee of dealership'
    .
    .
    .
class attributes:

    Number_of_cars_sold
        value class: INTEGERS
        derivation: number of unique members in
                        this class
        assertion: Number_of_cars_sold =
                    sizeof(SCHEDULED_PREPS) +
                    sizeof(PREPARED_CARS)
        failure action:
                    call _determine_cars_not_scheduled

Figure 4-1. Use of Member and Class Attribute and Interclass Assertions

Other known interclass connections which can be used to define the same class can also be stated within the interclass assertion. An example of the use of the interclass assertion is shown in Figure 4-2.

PREPARED_BUICKS
   description: { This class contains all prepared
                  Buick's for any dealership. }
   interclass connection: subclass of CARS where
                            is in BUICKS and
                            is in PREPARED_CARS
   interclass assertion: subclass of PREPARED_CARS where
                            Make = 'Buick'

Figure 4-2. Use of the Interclass Assertion as Multiple Interclass Connections

The failure action for the interclass assertion in Figure 4-2 is assumed to be the error action. If the interclass assertion were to fail, the database administrator would receive an error message indicating that an invalid database state has been entered.

4.2 Grouping Name Classes

In the real world, the value of one attribute, A_1, may determine the value class of another attribute, A_2. An example of this is the attributes "Make" and "Model" of class CARS. Any combination of Make and Model of cars may not make sense. In fact, the make of a car determines the possible models which can be defined. That is, the value of name class CAR_TYPES *determines* the value class for the name class CAR_MODELS. Thus, the name class CAR_MODELS is actually a *grouping* of value classes, where each value class is made up of a list of possible car models.

Since the value of an attribute A_1 belonging to name class C_1 determines the value class, VC_i, of an attribute A_2 belonging to name class C_2, the number of value classes for class C_2 is exactly the same as the number of members of name class C_1. This is because each value of name class C_1 determines one and only one value class VC_i belonging to name class C_2.

How this extension would be represented in the SDM schema is shown in Figure

- 41 -

4-3.

CAR_TYPES

    .
    .
    .

    determines: CAR_MODELS

CAR_MODELS

    .
    .
    .

    interclass connection: grouping of STRINGS as specified
                      by CAR_TYPES

Figure 4-3. Example of Grouping Name Class Usage

In Figure 4-3, the term "determines" means that a value of the given name class (e.g., CAR_TYPES) determines the value class of the other specified name class (e.g., CAR_MODELS). Logically, the CAR_MODELS name class is thought of as being a class which has members which are name classes (as in the grouping subclass definition). The interclass connection for the CAR_MODELS name class would then specify that a "grouping" of name classes exist and then specify the predicate, p, which is used to formulate each name class in the group. Finally, the grouping name class will indicate the name class (e.g., CAR_TYPES) which determines which value class of the group will be selected. This defines a symmetrical relationship between the name class "determining" the second and the name class being "determined by" the first.

4.3 Constant Name Class Members

Constant name class members are a method of identifying a permanent member of a given name class which can then be used throughout the rest of the SDM schema. Constant name class members are most beneficial when used with the assertion clause presented in section 3.1. An example of constant name class members might

be CURRENT_DATE of name class DATES and CURRENT_YEAR of name class YEARS. Typically, constant name class members would be values which would be assigned by the database administrator or some other external source.

Constant members would be declared within the name class which it is defined as follows:

C
    definition: ...
    interclass connection: ...
    constant members: $M_1, M_2, ..., M_n$

where the value of $M_1, M_2, ..., M_n$ would be automatically defined as the value class of C.

4.4 Sizeof Class Operator

The *sizeof* class operator is supplied to add to the capabilities of defining member and class attribute assertions. Simply, the sizeof operator defines the *size of* the class C. That is,

sizeof(C) $\equiv$ number of members in class C.

Figure 4-1 demonstrates the use of the sizeof class operator.

4.5 Summary

This chapter discusses some entensions to the SDM which are included in the SDML grammar as described in Chapter 4. Appendix 2 and 3 give the structure of the SDML specification which will be the foundation for the development of the SDML grammar in Chapter 4. All of the examples used in this chapter are included in the SDML specification shown in Appendix 5 (which is an upgrade of the example in Appendix 1).

The following items are identified as warranting further study:

- Automatic database recovery specifications in SDML,

- Procedure specifications in SDML,

- Further combining some of the derivation predicates to support a smaller subset of possible predicates while retaining current capabilities.

- On a larger scale, the SDM schema should be re-worked to provide a more cohesive set of capabilities instead of seamingly ad-hoc capabilities.

Chapter 4 will give a brief introduction to formal language theory and begin describing the design of the SDML grammar.

Chapter 4: SDML Grammar Definitions

Before presenting the design of the SDML grammar, a short discussion on languages and grammars is beneficial. Afterwards, the notation used in the rest of this chapter is described. Finally, the grammar for SDML will be presented.

5.1 Languages and Grammars

A *language* is an ordering of symbols based on some pre-defined set of rules. The symbols of a language constitute its *alphabet*. Symbols of an alphabet are referred to as *tokens* in compiler design. The rules which govern how the symbols of the alphabet may be arranged in the language is called the *grammar* for the language. Thus, given an alphabet, Σ, the following holds for the language L:

$$L \subset \Sigma^*.$$

where Σ^* is defined as the set of all *sentences* formed by concatenation of the tokens from Σ (including the *null* sentence).

A *terminal* symbol is any member of Σ. Therefore, a terminal symbol is synonymous with a token. A *nonterminal* symbol represents a substring of a sentence in L. Nonterminal symbols are used in a grammar as a building block for sentence construction. The set of terminal symbols, Σ, and nonterminal symbols, N, are *disjoint*; that is

$$\Sigma \bigcap N = \Phi.$$

The *production rules*, P, of a grammar, Γ, are a set of rules which define how the terminal symbols (or tokens) are built to form the language, L. A production rule has the following form:

$$\alpha \rightarrow \beta.$$

where $\alpha, \beta \in (N \bigcup \Sigma)^*$. In other words, a production rule defines a derivation step where a string of terminal and nonterminal symbols (α) derive another string of terminal and nonterminal symbols (β). A grammar Γ is then defined to be the four-tuple

$$\Gamma \equiv (\Sigma, N, P, S_s).$$

where S_s is the start symbol for the grammar Γ. The start symbol is a special nonterminal symbol such that, through one or more iterations of production rules from Γ, it will derive all sentences, s, in L. That is,

$$S_s \Longrightarrow^+ s, \text{ for } any \ s \in L(\Gamma).$$

We can therefore define the language defined by grammar Γ to be as follows

$$L(\Gamma) \equiv \left\{ s \mid s \in \Sigma^* \text{ and } S_s \Longrightarrow^+ s \right\}.$$

where $S_s \Longrightarrow^+ s$ means that s can be derived from the start symbol by application of one or more production rules, P, from Γ.

Placing restrictions on the form of production rules in a grammar Γ produces several well known classes of grammars. A grammar Γ is *context-sensitive* if

a. α contains at least one nonterminal symbol

b. $|\alpha| \leqslant |\beta|$

The above restrictions imply that a context-sensitive grammar is also $\epsilon-free$; that is, no productions of the form $\alpha \rightarrow \epsilon$ exist in the grammar, where ϵ is called the *empty* symbol. A grammar Γ is defined as a *context-free* grammar if the left hand side (LHS) of every production rule contains one and only one nonterminal

symbol. Then the form of each production in a context-free grammar is

$X \rightarrow \beta.$

where $X \in N$ and $\beta \in (N \bigcup \Sigma)^*$. Notice that the right hand side (RHS) of a context-free grammar rule can be the empty string ϵ. This implies that context-free grammars need not be context-sensitive. A *left-recursive* grammar is a context-free grammar which includes productions of the form

$X \rightarrow X\nu.$

where $X \in N$ and $\nu \in (N \bigcup \Sigma)^*$.

The SDML grammar will be designed to be context-free, allowing empty symbols on the RHS of rules. The SDML grammar will, therefore, not be context-sensitive. The SDML grammar will be defined as a left-recursive grammar to implement list structures. Reasons for left-recursion (vs. right-recursion) will be discussed in Chapter 5.

It is desirable a grammar to be *unambiguous*; that is, for each sentence, $s \in L$, there exists a unique parse tree for s defined by Γ. When a ambiguous grammar is parsed, the parser must anticipate the intended meaning of the ambiguous sentence and "guess" at a derivation for the sentence. It is possible that, if the parser guesses incorrectly, the sentence cannot be fully reduced given the derivation chosen by the parser. Since it has been proven that there exists no algorithm which can take an arbitrary context-free grammar and prove that it is ambiguous or not [10][11], a general statement about the ambiguity of the SDML grammar presented cannot be made here. Therefore, the question of SDML grammar ambiguity will be left to Chapter 5 (where the SDML parser will be discussed). Note, however, that some automatic parser generators (including the one chosen for the SDML grammar) do

supply *disambiguating rules* for resolving ambiguities in the grammar.

The SDML grammar will be developed as an LR(1) grammar. The primary reason for choosing an LR(1) grammar for SDML is to allow the use of already existing parser generators for parsing an SDML specification. An LR(1) grammar is one which can be parsed *L*eft-to-right producing a *R*ightmost derivation in reverse, with one token look ahead. Barrett and Couch [10] give the following definition for an LR(1) grammar. Consider a grammar Γ with start symbol S_r and a rightmost derivation of a terminal string w as follows:

$$S_r \;=> \; w_1 \;=> \; w_2 \;=> \; \cdots \;=> \; w \, .$$

Now consider a typical step in the derivation:

$$\alpha A t \;=> \; \alpha \beta t \, .$$

where $A \rightarrow \beta$ is the production used to reduce $\alpha A t$ to $\alpha \beta t$, and $\alpha \beta t$ is one of the w_i. Then Γ is LR(1) if, for every such derivation step, the production $A \rightarrow \beta$ can be inferred by scanning $\alpha \beta$ and the first token of v. In this definition, $t \in \Sigma^*$, $A \in N$, and $\alpha, \beta \in (\Sigma \bigcup N)^*$. Developing an LR(1) grammar for SDML will allow for the design of a shift-reduce parser for SDML with a one token look ahead.

Notice that an LR(1) grammar provides an extremely powerful language specification. That is, the amount of information used to infer a reduction can be immense. That is, the entire stack of shifted symbols (i.e., $\alpha \beta$) as well as the first token of v, is used to infer the reduction by the production $A \rightarrow \beta$. How the parser actually implements this is left to Chapter 5.

5.2 Notation

In the SDML grammar specification described in the next section, the following notation is used:

1. Nonterminal symbols are lower-case characters enclosed within the $<$ and $>$ symbols (e.g., $<$class-attributes$>$).

2. Terminal symbols are denoted as upper-case strings,

3. If two productions (e.g., $X \rightarrow Y$ and $X \rightarrow Z$) have the same LHS, they are shown as

$$X ::= Y \mid$$
$$\quad\quad Z,$$

where the '$\mid$' symbol represents an alternative. A production of the above form represents the Backus-Naur Form (BNF) for the grammar, Γ. The SDML grammar will be presented in the BNF format.

4. When referring to a mapping, the terminology AM_1 will be used (read Attribute Mapping sub 1). When referring to a mapping list, AM_k will be used.

5. Defining the attribute mapping AM_1 as a mapping of attributes N_i (i.e., $N_1 N_2 \cdots N_m$), then attribute N_i in the mapping sequence within AM_1 is referenced as $(AM_1)^{N_i}$.

6. The value class for an attribute or mapping (say AM_i) will be denoted by VC_{AM_i} (read "the value class of AM_i").

7. The underlying base class for class C_i is denoted as U_{C_i} (read "the underlying base class of C_i").

Class names, attribute names, procedure names, constant member names, and labels are considered as terminal symbols in the discussion since they are passed from the lexical analyzer as such.

5.3 SDML Grammar Specification

This section will discuss the development of the grammar for SDML. With each production sequence, the semantic checks which will be performed by the SDML parser are listed. All production rules are listed in Appendix 4 as they were given to the Yacc program. The order of the discussion follows the order of the productions given in Appendix 4.

The following checks will be made in reference to class and attribute name definitions:

a. class name definitions must be unique for all classes defined in the SDML specification.

b. attribute name definitions must be unique for the underlying base class, U, and all eventual subclasses of U.

c. constant member name definitions must be unique with respect to all constant member names defined in the SDML specification.

As discussed in Chapter 2, the SDM schema is made up of one or more class definitions. Thus, the SDML grammar start symbol will be <sdm-schema> and will, therefore, be defined as a class definition list as follows:

```
<sdm-schema>  ::=  <class-definition-list>
```

A class definition list can be either a single class or multiple class definition lists. In either case, a first (and last) class definition must be present. The <class-definition-list> nonterminal could then consist of a single class definition or a last class definition preceded by other class definitions (i.e., another class definition list):

```
<class-definition-list>  ::=  <class-definition>  |
                  <class-definition-list>  <class-definition>
```

Base and nonbase class definitions can be mixed withing the SDML specification.

As discussed in Chapter 2, a class in the SDM schema is either defined as as base class or a nonbase class. The distinction between base and nonbase classes is not declared explicitly in the SDM schema with the class name, but is implicit in how the class is defined. A distinction can be made between base classes and nonbase classes by the presense of an *interclass connection*. All nonbase classes have interclass connections and no base classes have interclass connections. Therefore, no conflict should arise in the grammar if the class definition produces either a base class definition or a nonbase class definition as follows:

```
<class-definition>  ::=  <base-class-definition>  |
                  <nonbase-class-def>
```

Let us first concentrate on the base class definition. Every class, whether base or nonbase, is identified with a unique class name possible followed by multiple synonymous names. The class name list is then followed by the body of the base class as follows:

```
<base-class-definition>  ::=  <class-name-list>  <base_class_body>
```

The class name list is headed by the *primary name* for the class optionally

followed by the list of synonymous names separated by comma's or and's:

```
<class-name-list> ::= CLASS_NAME |
                <class-name-list> <comma-and-sep> CLASS_NAME

<comma-and-sep> ::= ',' | AND
```

Within the list of semantic checks for declarations within the body of this class, the primary name of the class presently being defined will be C.

Now for the definition of the body of a base class. A base class is made up of an optional description, an optional base class features declaration, one or more member attributes, zero or more class attributes, and a list of identifiers. Thus, the body of a base class is as follows:

```
<base_class_body> ::= <desc-clause> <bc-feat-decl>
                <member-attr-decl> <class-attr-decl>
                <ident-decl>
```

The description clause allows the designer to provide an Engligh language sentence for documentation purposes. The description clause has the following format:

```
<desc-clause> ::= ε |
            DESCRIPTION : DESCRIPTION_TEXT
```

Base class features allow the designer to specify whether or not duplicates are allowed within this class. If not specified, the default is that duplicates are allowed. The base class features declaration is then

```
<bc-feat-decl> ::= ε |
            DUPLICATES NOT ALLOWED |
            DUPLICATES ALLOWED
```

Every base class must have one or more member attributes defined for the class. Thus the productions

```
<member-attr-decl>   ::=  MEMBER ATTRIBUTES : <member-attr-list>

<member-attr-list>   ::=  <member-attribute>  |
                          <member-attr-list> <member-attribute>
```

define the format of member attribute definitions within a class. Member attributes are identified by an attribute name optionally followed by multiple synonymous attribute names. Member attribute names must be unique within all attributes (member and class) of this class and all eventual subclasses of this class. The body of a member attribute contains the description clause (exactly as in the class definition), the value class declaration for the attribute, an optional inverse relationship, an optional match or derivation, a member order defining the order of the attribute, and member attribute options. Added to the member attribute definition is the member attribute assertion. Thus, the production defining a member attribute definition is as follows:

```
<member-attribute>  ::=  <attr-name-list> <desc-clause>
                         <value-class-decl> <inverse-decl>
                         <match-or-derivation> <member-order>
                         <member-attr-opts> <attr-assertion-decl>

<attr-name-list>    ::=  ATTRIBUTE_NAME |
                         <attr-name-list> comma-and-sep ATTRIBUTE_NAME
```

The primary name for the attribute currently being built will be A (within class C).

The value class declaration defines the value class from which the attribute will derive its value. The value class declaration is as follows:

```
<value-class-decl>  ::= VALUE CLASS : CLASS_NAME
```

The value class definition for A will be denoted VC_A.

The inverse declaration allows the designer to define an inverse attribute

interrelationship between this attribute and another. An inverse relationship logically defines an assertion between this attribute and the inverse attribute. Therefore, any additional assertions will be redundant. Recall from Chapter 2 that the inverse attribute must have a value class of the class presently being defined and this attribute must have a value class of the inverse attributes underlying class. These checks, however, will not be made here since the inverse attribute may not have been defined yet. The inverse checks will be made at the end of the first pass after all classes and attributes have been discovered and appropriate symbol tables have been built. The inverse declaration is defined as follows:

    <inverse-decl> ::= INVERSE : ATTRIBUTE_NAME

The semantic checks which will be performed on the <inverse-decl> production are as follows:

  Production:

        <inverse-decl> $\longrightarrow$ inverse : A_1

  Checks:

          i. A_1 is defined as a member attribute within VC_A

          ii. A_1 has a value class of C

          iii. A_1 has "inverse : A" definition

As well as an inverse declaration, the member attribute can contain either a match or derivation (but not both). Therefore, the production

    <match-or-derivation> ::= <match-decl> |
               <derivation-decl>

will provide the exclusive-or capability. Any member attribute definition rules will be applied to the class definition after pass 1 when all required data is available to perform the checks.

The match attribute interrelationship is defined by the following production:

<match-decl> ::= MATCH : ATTRIBUTE_NAME OF CLASS_NAME
ON ATTRIBUTE_NAME

All checks on attribute value classes and underlying classes noted in Chapter 2 will be done after pass 1 of the parser when all required data is available. These semantic checks are as follows:

Production:

<match-decl> $\longrightarrow$ match: A_1 of C_1 on A_2

Checks:

    i. A_1 and A_2 must be defined as member attributes within C_1

    ii. A_1 has a value class of VC_A

    iii. A_2 has a value class of C.

    iv. if A has an inverse declaration (call it A_3), then

        a. A_3 has no derivation declaration.

        b. if A_3 has a match declaration then it must be:

            match: A_2 of C_1 on A_1.

The derivation attribute interrelationship is defined by the following production:

<derivation-decl> ::= DERIVATION : <member-attr-deriv>

where the <member-attr-deriv> defines all legal attribute derivations for member attributes. The following semantic checks will be make for the given production:

Production:

<derivation-decl> $\longrightarrow$ derivation: <member-attr-deriv>

Checks:

    i. if A has an inverse declaration (call it A_3) then:

        a. A_3 has no derivation declaration.

        b. A_3 has no match declaration.

Member attribute derivations will be discussed later in this chapter.

The member order for a member (or a class) attribute defines the member (or class) attribute to be either single valued or multivalued. Multivalued attributes can also have a restriction on the number size of the attribute (i.e., array size). If member order is not specified, then single valued is assumed. The production for member order is as follows:

```
<member-order>  ::= ϵ |
            SINGLE VALUED |
            MULTIVALUED |
            MULTIVALUED WITH SIZE BETWEEN |
            INTEGER_C AND INTEGER_C
```

Note that the bounds on the array size must be specified as being between two integer constants.

Member options allow the designer to specify one of four possible attribute options. The phrase "may not be null" may be specified which indicates that the attribute cannot contain the special value *null*. The attribute can be declared as "not changeable" which implies that once a non-null value is given to the attribute, it cannot be changed. The "exhausts value class" phrase within the definition of attribute A for class C indicates that for every value V in the value class for the attribute A, there must exist at least one member M_i of the underlying class C such that the attribute A value for that member M_i is that value V. The "no overlap in values" phrase indicates that if some member M_i of class C has contains a value V for attribute A, then there cannot exist another member M_j such that the attribute A value of member M_j is contains V. The "no overlap in values" option is only valid for multivalued member attributes since single valued

attributes are nonoverlapping by definition. Any or none of the above mentioned options can be specified for a member attribute. The BNF sequence for definition of member attribute options is thus:

```
<member-attr-opts>  ::=  <member-options>  |  ε

<member-options>  ::=  <member-opt-item>
                       <member-options>  <member-opt-item>

<member-opt-item>  ::=  MAY NOT BE NULL  |
                        NOT CHANGEABLE  |
                        EXHAUSTS VALUE CLASS  |
                        NO OVERLAP IN VALUES
```

The reason for designing the grammar to allow a list of the available options is to allow the options to be entered in any order. This does, however, allow a single option to be entered more than once within the same member attribute definition. The parser can detect this at the time of the parsing and simply reduce the redundant definitions to a single definition. The following semantic checks are made for the indicated member options:

Production:

```
<member-opt-item>  ——→  exhausts value class
<member-opt-item>  ——→  no overlap in values
```

Checks:

1. A is defined as a multivalued attribute.

The member attribute assertion extension allows some statement of fact to be made about this member attribute, other member attributes within this class, class attributes within this class, or constants. Mappings can be used when referencing member attributes within this class to reference lower level data values. A member attribute assertion is made up of one or more assertion statements followed by a failure action if any one of the assertions were to fail. Therefore, the productions involving declaration of a member attribute assertion are as

follows:

```
<attr-assertion-decl> ::= ε |
            <attr-assertion-list> <fail-action-clause>

<attr-assertion-list> ::= <attribute-assertion> |
            <attr-assertion-list> <attribute-assertion>

<attribute-assertion> ::= ASSERTION : <assertion>
```

One of three possible assertions can be used in the member attribute assertion:

1. A call to a user-defined procedure which will provide the necessary data checks to verify the current database state.

2. A comparison of two arithmetic expressions involving member attributes and class attributes within this class and constants. With this type of member attribute assertion, each of the arithmetic expressions must evaluate to a number value (INTEGER_C or REAL_C). The comparison can be made with any one of the relational operators.

3. A statement indicating a set relationship between this member attribute and one of: (1) some other attribute or mapping within this class, (2) a class attribute within this class, or (3) another class. The two set operators which can ge used (defined later) are the "is contained in" and the "contains" operators. When the "is contained in" operator is used, the RHS of the comparison must be a multivalued attribute or a class. When the "contains" operator is used, the LHS of the comparison (representing the attribute being defined) must be a multivalued attribute.

The productions which define the member attribute assertion is then as follows:

```
<assertion>  ::=  CALL PROCEDURE_NAME |
         <mapping-expression> <relop> <mapping-expression> |
         <setop-predicate>
```

With a member attribute assertion list, an optional failure action can be supplied which indicates the action to take when the member attribute assertion fails. One of three possible failure actions are possible:

1. Call a user-defined procedure which will handle the error recovery from the invalid database state. Note that the user-defined procedure is then responsible for re-checking the new database state if an attempt was made to correct the error.

2. Flag the database state as an error state and report the error to the database administrator. An optional action message can be supplied to supply additional information to the database administrator. The action message is defined later in the SDML grammar.

3. Warn the database administrator of the condition with a message. This assertion may not actually represent an invalid database state, but might be used to notify the database administrator when the member attribute takes on some specified value(s).

The productions involved in specifying the failure action for a member attribute assertion are then:

```
<fail-action-clause>  ::=  ∈ |
              FAILURE ACTION : <failure-action>

<failure-action>  ::=  CALL PROCEDURE_NAME |
           ERROR <action-message> |
           WARNING STRING_C
```

The optional class attributes declaration section of a class definition can contain one or more class attribute definitions. Thus the productions

```
<class-attr-decl>  ::= ε |
                   CLASS ATTRIBUTES : <class-attr-list>

<class-attr-list>  ::= <class-attributes>
                   <class-attr-list> <class-attributes>
```

begin the definition of the class attribute declaration section.

Any given class attribute is uniquely identified with the attribute name. Class attributes can also be given multiple synonymous names. As with member attribute names, the class names must be unique within the underlying base class and all eventual subclasses of the base class. Class attributes contain an optional description clause, a value class declaration, an optional class derivation declaration, member order definition, class attribute options, and an optional class attribute assertion. The definition of the class attribute is then

```
<class-attribute>  ::= <attr-name-list> <desc-clause>
                   <value-class-decl> <class-deriv-decl>
                   <member-order> <class-attr-opts>
                   <attr-assertion-decl>
```

where the $<$attr-name-list$>$, $<$desc-clause$>$, $<$value-class-decl$>$, $<$member-order$>$, and $<$attr-assertion-decl$>$ are defined in exactly the same way as for the member attribute definition. The class attribute options, however, are either "may not be null" or "not changeable". The "exhausts value class" and the "no overlap in values" options do not make sense here since there is only a single instance of the class attribute independent of the number of members of the class. The productions for the class attribute options are

```
<class-attr-opts>  ::=  <class-options>  |  ε

<class-options>  ::=  <class-opt-item>  |
                      <class-options>  <class-opt-item>

<class-opt-item>  ::=  MAY NOT BE NULL  |
                       NOT CHANGEABLE
```

The optional class derivation declaration is specified as follows:

```
<class-deriv-decl>  ::=  ε  |
                         DERIVATION : <class-attr-deriv>
```

where <class-attr-deriv> defines the valid derivation primitives which can be used for class attributes.

Let us first begin the discussion of derivation primitives with the member attribute derivation since it was introduced first in the SDML grammar. A member attribute derivation can be either an interattribute derivation or a member-specific derivation. Interattribute derivations are those which can appear in either the member attribute derivation or the class attribute derivation. Thus, the productions for the member attribute derivation are

```
<member-attr-deriv>  ::=  <interattr-deriv>  |
                          <member-spec-deriv>
```

An interattribute derivation used within a member attribute derivation requires the following semantic check(s) be made:

Production:

$$<member\text{-}attr\text{-}deriv> \longrightarrow <interattr\text{-}deriv>$$

Checks:

i. $(AM_i)^{N_i}$ must be defined as a member attribute within class C.

Member-specific derivations are derived using one of four predicates: (1) ordering predicate, (2) existence predicate, (3) recursive trace predicate, or (4) contents

predicate. The member-specific derivation is then defined as follows:

```
<member-spec-deriv>  ::=  <ordering-predicate>  |
                          <existence-predicate>  |
                          <recursive-trace-pred>
```

The ordering predicate defines the member attribute to be the sequential position of the member when ordered by some specified attribute(s). The ordering can be either increasing or decreasing (default is increasing). The ordering predicate can also define the attribute to be the sequential position of the member when ordered by some specified attribute(s) grouped on a common value of some other specified attribute(s) by using the within clause. Since the attribute has a value representing the sequential position of the member within some list, the ordering-predicate requires the value class of the attribute to be an integer constant (value class INTEGERS). The productions which define an ordering predicate are

```
<ordering-predicate>  ::=  ORDER BY  <direction>
                           <mapping-list>  <within-clause>

<direction>  ::=  INCREASING  |  DECREASING  |  ε

<within-clause>  ::=  WITHIN  <mapping-list>  |  ε

<mapping-list>  ::=  <mapping>  |
                     <mapping-list>  <comma-and-sep>  <mapping>
```

The productions from above with their checks are listed below:

Production:

$$<\text{ordering-predicate}> \ \longrightarrow \ \text{order by } AM_k \ <\text{within-clause}>$$

Checks:

    i. $VC_A \equiv \text{INTEGERS}$

    ii. $(AM_i)^{N_1}$ must be defined in C for all i in k.

Production:

$$<\text{within-clause}> \longrightarrow \text{within } AM_j$$

Checks:

    i. $(AM_j)^{N_1}$ must be defined in C for all j.

The existence predicate defines an attribute to be either "true" or "false" based on whether the member is a member of some specified class. The production

    $<\text{existence-predicate}>$ ::= IF IN CLASS_NAME

defines an existence predicate. Since the existence predicate defines a yes/no alternative, the value class of the defining attribute must be the built-in class YES/NO. The checks for the existence predicate are as follows:

Production:

$$<\text{existence-predicate}> \longrightarrow \text{if in } C_1$$

Checks:

    i. $C_1 \equiv$ YES/NO

    ii. $U_C \equiv U_{C_1}$

The recursive trace predicate defines the value of the attribute A to be all members which are found by recursively tracing the values of some specified attribute A_1 of the same class C. In order for the recursive trace predicate to make sense, both attributes A and A_1 must have a value class of the underlying class C. A limit can also be placed on the level of the recursive trace. The productions which define the recursive trace predicate are

    $<\text{recursive-trace-pred}>$ ::= $<\text{level-clause}>$ LEVELS OF VALUES
                                      OF ATTRIBUTE_NAME

    $<\text{level-clause}>$ ::= UP TO INTEGER_C | ALL

The semantic checks for the recursive trace predicate are as follows

Production:

$$\text{<recursive-trace-pred>} \longrightarrow \text{<level-clause> levels of values of } A_1$$

Checks:

    i. $VC_A \equiv C$,

    ii. $VC_{A_1} \equiv C$,

    iii. A_1 is defined as a member attribute attribute within class C.

Next, the class attribute derivation will be discussed. The class attribute derivation can be either an interattribute derivation or a class-specific derivation. The production for the class attribute derivation is then

    <class-attr-deriv> ::= <interattr-deriv> |
                 <class-spec-deriv>

An interattribute derivation used within the class attribute derivation requires the following semantic check:

  Production:

    <class-attr-deriv> $\longrightarrow$ <interattr-deriv>

  Checks:

    i. $(AM_i)^{N_1}$ must be defined as a class attribute within class C.

First, the class specific derivation will be discussed followed by the interattribute derivation (which can be used by either the class attribute derivation or the member attribute derivation). Two types of class-specific derivation predicates are allowable: the class size predicate and the class member predicate:

    <class-spec-deriv> ::= <class-size-pred> |
                 <class-member-pred>

The class size predicate defines the attribute to be the number of members in the underlying class. The class member predicate defines the attribute to be the sum, average, minimum, or maximum of some member attribute taken over all members

- 64 -

of the underlying class. The following productions define the class-specific derivation predicates:

```
<class-size-pred>  ::= NUMBER OF <uniqueness> MEMBERS
                       IN THIS CLASS

<uniqueness>  ::= UNIQUE | ε

<class-member-pred>  ::= <set-function> OF ATTRIBUTE_NAME
                         OVER MEMBERS OF THIS CLASS

<set-function>  ::= MINIMUM | MAXIMUM | AVERAGE | SUM
```

The following semantic checks must be made within the following productions from above:

Production:

> $<$class-size-pred$>$ $\longrightarrow$ number of $<$uniqueness$>$ members in this class

Checks:

    i. $VC_A \equiv$ INTEGERS.

Production:

> $<$class-member-pred$>$ $\longrightarrow$ $<$set-function$>$ of A_1 over members of this class

Checks:

    i. A_1 must be defined as member attribute within class C.

    ii. Given C_1 is the value class of A, $U_{C_1} \equiv$ NUMBERS, REALS, or INTEGERS.

    iii. Given $C_2 \equiv$ value class of A_1, $U_{C_2} \equiv$ NUMBERS, REALS, or INTEGERS.

The interattribute derivation can be specified using one of four predicates: (1) set-derived predicate, (2) equality predicate, (3) set-order predicate, or (4) subvalue predicate:

```
<interattr-deriv>  ::=  <derived-predicate>  |
                        <set-derived-pred>  |
                        <equality-predicate>  |
                        <set-order-predicate>  |
                        <subvalue-predicate>
```

The derived predicate specifies that the attribute A is directly derived from another attribute mapping. The production

```
<derived-predicate>  ::=  SAME AS  <mapping>
```

defines a derived predicate. The semantic checks required for the derived predicate are as follows:

Production:

$$<\text{derived-predicate}> \longrightarrow \text{same as } AM_1$$

Checks:

    i. $(AM_1)^{N_1}$ is defined in C

    ii. VC_{AM_1} (i.e., value class of $(AM_1)^{N_m}$) $\equiv VC_A$

The set-derived predicate defines the members of the attribute to be those members which are either in the union of two attribute mappings, in the intersection of two attribute mappings, or in the difference of two attribute mappings. The production

```
<set-derived-pred>  ::=  WHERE IS IN  <mapping>  AND IS IN  <mapping>  |
                         WHERE IS IN  <mapping>  OR IS IN  <mapping>  |
                         WHERE IS IN  <mapping>  AND IS NOT IN  <mapping>
```

defines the set-derived predicates for the interattribute derivations. The following semantic checks will be performed for the set-derived predicate:

Production:

$$<\text{set-derived-pred}> \longrightarrow \text{where is in } AM_1 \text{ and is in } AM_2$$
$$<\text{set-derived-pred}> \longrightarrow \text{where is in } AM_1 \text{ or is in } AM_2$$
$$<\text{set-derived-pred}> \longrightarrow \text{where is in } AM_1 \text{ and is not in } AM_2$$

Checks:

i. both $(AM_1)^{N_1}$ and $(AM_2)^{N_1}$ must be defined appropriately within class C

ii. Given C_1 is the value class of AM_1 and C_2 is the value class of AM_2, then $U_{C_1} \equiv VC_A$ and $U_{C_2} \equiv VC_A$

iii. AM_1 and AM_2 must be multivalued attribute mappings

The equality predicate defines the attribute to be directly derived from an arithmetic expression involving the following:

- set functions on multivalued member attributes,

- other member attributes,

- number constants,

- constant member names,

- size of multivalued member attributes, or

- size of classes.

Thus the series of productions which define an equality predicate are as follows:

<equality-predicate> ::= EQ <mapping-expression>

<mapping-expression> ::= <mapping-term> |
    <mapping-expression> <addition-operator> <mapping-term>

<mapping-term> ::= <mapping-factor> |
    <mapping-term> <multiply-operator> <mapping-factor>

<mapping-factor> ::= <mapping-primary> |
    <mapping-factor> <exponent-operator> <mapping-primary>

<mapping-primary> ::= (<mapping-expression>) |
        <set-function> (<mapping>) |
        <mapping> |
        <number> |
        CONST_MEMBER_NAME |
        SIZEOF (<mapping>) |
        SIZEOF (CLASS_NAME)

<addition-operator> ::= + | -

<multiply-operator> ::= * | /

<exponent-operator> ::= !

The following semantic checks will be made for the productions withing the equality predicate:

Production:

    <equality-predicate> $\longrightarrow$ = <mapping-expression>

Checks:

    i. for all mappings AM_i in <mapping-expression>, $(AM_i)^{N_1}$ must be appropriately defined within C

    ii. $U_{VC_A} \equiv$ NUMBERS, REALS, or INTEGERS

Production:

    <mapping-primary> $\longrightarrow$ <set-function> (AM_1)

Checks:

    i. Given $C_1 \equiv VC_{AM_1}$, $U_{C_1} \equiv$ NUMBERS, REALS, or INTEGERS

    ii. AM_1 must be a multivalued attribute or mapping

Production:

$$\langle \text{mapping-primary} \rangle \;\; \longrightarrow \;\; AM_1$$
$$\langle \text{mapping-primary} \rangle \;\; \longrightarrow \;\; \text{sizeof} \,(\, AM_1 \,)$$

Checks:

     i. AM_1 must be a multivalued attribute or mapping

Production:

$$\langle \text{mapping-primary} \rangle \;\; \longrightarrow \;\; \text{CONST\_MEMBER\_NAME}$$

Checks:

     i.   CONST_MEMBER_NAME must be defined within a name class C_1 such that $U_{C_1} \equiv$ NUMBERS, REALS, or INTEGERS

The set-order predicate defines the value of the attribute to be the number of members in some specified eventual attribute of a mapping. The production

$$\langle \text{set-order-predicate} \rangle \;\; ::= \;\; \text{NUMBER OF} \; \langle \text{uniqueness} \rangle$$
$$\text{MEMBERS IN} \; \langle \text{mapping} \rangle$$

defines the set-order predicate. The semantic checks which will be performed for the set-order predicate are as follows:

Production:

$$\langle \text{set-order-predicate} \rangle \;\; \longrightarrow \;\; \text{number of} \; \langle \text{uniqueness} \rangle \; \text{members in} \; AM_1$$

Checks:

     i. $(AM_1)^{N_1}$ is defined appropriately within class C

     ii. AM_1 is a multivalued attribute or mapping

     iii. $VC_A \equiv$ INTEGERS

The subvalue predicate defines the attribute A to be a subvalue of another mapping A_1 which satisfies some condition. The conditions can be either (1) values of A_1 which are members of some class C_1, or (2) values of A_1 which satisfy some attribute predicate. The attribute predicate will be described later in this chapter. All mappings, C_1, and A_1 must be the value class of A (i.e., must be VC_A). The

productions which form the subvalue predicate are:

```
<subvalue-predicate>  ::=  SUBVALUE OF <mapping> WHERE
                          <subvalue-selection>

<subvalue-selection>  ::=  IS IN CLASS_NAME |
                          <attribute-predicate>
```

The semantic checks which will be made for the above productions are as follows:

Production:

$$<\text{subvalue-predicate}> \longrightarrow \text{subvalue of } AM_1 \text{ where } <\text{subvalue-selection}>$$

Checks:

    i. $(AM_1)^{N_1}$ is defined appropriately within class C

    ii. AM_1 is a multivalued attribute or mapping

    iii. $VC_{AM_1} \equiv VC_A$

Production:

$$<\text{subvalue-selection}> \longrightarrow \text{is in } C_1$$

Checks:

    i.   given that $C_2 \equiv VC_{AM_1}$, then $U_{C_2} \equiv U_{C_1}$

Production:

$$<\text{subvalue-selection}> \longrightarrow <\text{attribute-predicate}>$$

Checks:

    i. $<\text{attribute-predicate}>$ must satisfy definition for class VC_A

A mapping is a sequential reference of attributes based on the value class of each attribute. Mappings are defined in the SDML grammar as follows:

```
<mapping>  ::=  ATTRIBUTE_NAME |
               <mapping> . ATTRIBUTE_NAME
```

Consider a mapping $N_1.N_2. \cdots .N_m$ where each N_i is defined within class C_i. In order for the mapping to make sense, the following must be true for any attribute

$_i$ within the mapping:

   i. $VC_{N_i} \equiv C_{i+1}$ for all $i = 1, 2, ..., m-1$

Finally, the last part of a base class definition is the base class identifiers. The base class identifiers can be

   i. no identifiers,

   ii. a single attribute identifier list separated by plus (+) signs, or

   iii. multiple attribute identifier lists separated by commas.

The productions defining the identifier declarations are then:

```
<ident-decl>  ::=  IDENTIFIERS : <ident-list>  |
             IDENTIFIERS : NONE

<ident-list>  ::=  <identifier>  |
             <ident-list> , <identifier>

<identifier>  ::=  ATTRIBUTE_NAME |
             <identifier> + ATTRIBUTE_NAME
```

The following semantic checks will be made for the identifier declaration:

Production:

          $<$ident-decl$>$ $-\!\!\!\rightarrow$ identifiers: $<$ident-list$>$

Checks:

          i. class C cannot specify that duplicates are allowed (since a unique member specification is given).

Production:

      $<$identifier$>$ $-\!\!\!\rightarrow$ A_1
      $<$identifier$>$ $-\!\!\!\rightarrow$ $<$identifier$>$ $+ A_1$

Checks:

i. A_1 must be defined as a member attribute within class C

The next topic for discussion is the construction of a nonbase class definition. As in the definition of the base class, nonbase classes are made up of a primary class name and possible synonymous names, followed by a nonbase class body.

```
<nonbase-class-def>  ::=  <class-name-list>
                          <nonbase-class-body>
```

The nonbase class body is made up of an optional description clause (like that of the base class), a mandatory interclass connection, and the nonbase class alternatives.

```
<nonbase-class-body>  ::=  <desc-clause>
                           <interclass-connection>
                           <nonbase-class-alts>

<nonbase-class-alts>  ::=  <nonbase-class-feat>  |
                           <name-class-feat>
```

The nonbase class alternatives will identify the nonbase class as a definite name class (if the determines clause or the constant members clause is specified) or a possible name class. A possible name class will be later identified as a name class if the class is specified as a subclass of one of the built-in classes and either the user-controllable subclass predicate or no subclass predicate is specified. A name class grouping predicate also identifies the nonbase class as a name class.

Nonbase class features allow non-name classes to be given member attributes and/or class attributes. Here (as opposed to base class definitions), nonbase class member attribute definitions are optional.

interclass connection specifies how this subclass derivation is determined. The interclass assertion is declared as follows:

```
<interclass-assertion> ::= INTERCLASS CONNECTION :
        <connection> <ic-assertion-decl>
```

where the connection establishes the interclass connection.

With the interclass connection, zero of more interclass assertions can be provided. The interclass assertion simply states additional interclass connections which must be satisfied by the nonbase class member list. With the interclass assertion list, a failure action can be specified. This failure action is the same as that of the attribute assertion failure action. The interclass assertion declaration is as follows:

```
<ic-assertion-decl> ::= ϵ |
        <ic-assertion-list> <failure-action-clause>

<ic-assertion-list> ::= <ic-assertion> |
        <ic-assertion-list> <ic-assertion>

<ic-assertion> ::= INTERCLASS ASSERTION : <connection>
```

An interclass connection can be either a subclass connection (specifying a subclass of the parent class) or a grouping connection (grouping the members of the parent class in some way).

```
<connection> ::= <subclass-connection> |
        <grouping-connection>
```

An important side effect of establishing a grouping connection is the automatic definition of the derived multivalued member attribute *Contents*, whose value class is the parent class. The "Contents" attribute has as its value for a given member of the grouping class, all members of the parent class which belong to one of the given groupings.

The first type of interclass connection is the subclass connection. The nonbase class is established as a subclass of the parent class by application of a subclass predicate on members of the parent class.

```
<subclass-connection>  ::= SUBCLASS OF CLASS_NAME
                <subclass-predicate>
```

Attributes and classes referenced in the <subclass-predicate> must satisfy semantic checks consistent with the definition of the parent class PC (see subclass predicate discussion below).

The second type of interclass connection is the grouping connection. A grouping connection can be established using any one of four methods: (1) expression defined grouping class, (2) enumerated grouping class, (3) user-controllable grouping class, or (4) name class grouping class.

```
<grouping-connection>  ::= <expression-defined-gc>  |
                <enumerated-gc>  |
                <user-controllable-gc>  |
                <name-class-gc>
```

The subclass predicate of the subclass connection allows definition of several types of subclasses: (1) attribute-defined subclass, (2) user-controllable subclass, (3) set-operator defined subclass, (4) existence subclass, or (5) name class subclass.

```
<subclass-predicate>  ::= WHERE <attribute-predicate>  |
                WHERE SPECIFIED  |
                WHERE <set-oper-defined-sc>  |
                WHERE <existence-sc>  |
                <name-class-sc>
```

Attribute mappings referenced in the <attribute-predicate> of the attribute-defined subclass must satisfy semantic checks consistent with the definition of the parent class PC. The attribute-defined subclass and the user-controllable subclass C will inherit all member attributes defined in the parent class PC.

The set–operator defined subclass defines the class C to be all members of the parent class PC which satisfies an intersection, union, or difference of two classes.

```
<set-oper-defined-sc>  ::=  IS IN CLASS_NAME AND
                            IS IN CLASS_NAME |
                            IS IN CLASS_NAME OR
                            IS IN CLASS_NAME |
                            IS NOT IN CLASS_NAME
```

The semantic checks which will be performed for the set–operator defined subclass are as follows:

Production:

$$<\text{set-oper-defined-sc}> \ -\rightarrow \ \text{is in } C_1 \text{ and is in } C_2$$
$$<\text{set-oper-defined-sc}> \ -\rightarrow \ \text{is in } C_1 \text{ or is in } C_2$$
$$<\text{set-oper-defined-sc}> \ -\rightarrow \ \text{is not in } C_1$$

Checks:

    i. $U_{C_1} \equiv U_{PC}$

    ii. $U_{C_2} \equiv U_{PC}$, if C_2 specified

The attribute inheritance rules for the set–operator defined subclass are as follows:

a. the intersection defined subclass C will inherit all member attributes common to C_1 and C_2

b. the union defined subclass C will inherit all member attributes of both C_1 and C_2

c. the difference defined subclass C will inherit all member attributes of the parent class PC

The existence subclass specifies the class C to be all members of the parent class PC which is currently a value of some member attribute A_1 of class C_1.

```
<existence-sc>  ::= IS A VALUE OF ATTRIBUTE_NAME
                OF CLASS_NAME
```

The existence subclass C will inherit all member attributes of its parent class PC. The semantic checks which will be performed for the existence subclass definition are as follows:

Production:

$<$existence-sc$>$ $\longrightarrow$ is a value of A_1 of C_1

Checks:

i. A_1 is defined as a member attribute within class C_1

ii. $VC_{A_1} \equiv PC$

A name class definition can either (a) not specify a subclass predicate, or (b) specify a format directive on class STRINGS.

```
<name-class-sc>  ::= ε |
                WHERE FORMAT IS <format-directive>
```

The parent class PC of a name class must be one of the built-in classes; therefore, no member attribute inheritance is involved. The semantic checks which will be performed on a name class specification are as follows:

Production:

$<$name-class-sc$>$ $\longrightarrow$ ε

Checks:

i. $PC \equiv$ STRINGS, NUMBERS, REALS, or INTEGERS

Production:

$<$name-class-sc$>$ $\longrightarrow$ where format is $<$format-directive$>$

Checks:

i. $PC \equiv$ STRINGS

The first type of grouping connection was the expression defined grouping connection. This type of grouping connection defines the class C to be a grouping of the parent class PC based on common values of one or more attributes of PC.

```
<expression-defined-gc>  ::=  GROUPING OF CLASS_NAME
                              ON COMMON VALUE OF
                              <attr-name-list>
                              <explicitly-def-grps>
```

```
<explicitly-def-grps>  ::=  GROUPS DEFINED AS CLASSES
                            ARE <class-name-list>
```

Any groups of members of the parent class PC which are also explicitly defined as attribute-defined subclasses are listed. These groupings represent redundant class definitions, one within the grouping defined and the other explicitly defined as a subclass of PC. The semantic checks which will be performed for the expression defined grouping connection are as follows:

Production:

<center><expression-defined-gc> $\longrightarrow$ grouping of PC on common value of A_k
<explicitly-def-grps></center>

Checks:

1. A_k must be defined as a member attribute of PC for all k

The next type of grouping connection is the enumerated grouping connection. The enumerated grouping connection defines the class C as a grouping of classes C_k.

```
<enumerated-gc>  ::=  GROUPING OF CLASS_NAME CONSISTING
                      OF CLASSES <class-name-list>
```

The semantic checks which will be performed for the grouping connection are as follows:

Production:

<center><enumerated-gc> $\longrightarrow$ grouping of PC consisting of classes C_k</center>

```
<nonbase-class-feat>  ::=  <nbc-member-attr-decl>
                      <class-attr-decl>

<nbc-member-attr-decl>  ::=  ε  |
                      <member-attr-decl>
```

As mentioned earlier, name class features include the determines clause and the constant member clause.

```
<name-class-feat>  ::=  <determines-clause>  |
      <const-members-clause>  |
      <determines-clause>  <const-members-clause>

<determines-clause>  ::=  determines: CLASS_NAME

<const-members-clause>  ::=  constant members:
                      <const-members-list>

<const-members-list>  ::=  CONST_MEMBER_NAME  |
      <const-members-list>  <comma-and-sep>
                      CONST_MEMBER_NAME
```

The determines clause allows the dependence of the value class of the specified class name upon a value of the given name class. That is, a value of this name class *determines* the value class of the specified name class. The semantic checks which will be performed for the above productions follows:

Production:

$$<determines\text{-}clause> \; -\rightarrow \; determines: C_1$$

Checks:

i. the name class definition for C_1 must be defined as a grouping name class "where specified by C"

ii. C must contain the user-controllable interclass connection

The interclass connection is what distinguishes a base class from a nonbase class in the SDM schema. The members of a nonbase class are eventually derived from the members of the underlying base class. This derivation could be directly from a base class or indirectly through a series of nonbase subclass definitions. The

Checks:

$$\text{i.} \quad U_{C_k} \equiv U_{PC} \text{ for all } k$$

The user-controllable grouping class allows the user to specify the grouping of the parent class PC.

```
<user-controllable-gc>  ::= GROUPING OF CLASS_NAME
                            AS SPECIFIED
```

No semantic checks are required for the user-controllable grouping connection.

The name class grouping connection allows the database designer to specify the name class as a grouping of value class definitions; the value class group being determined by the value of another class C_1. Both C_1 and C must be user-controllable connections to keep the grouping connection at a finite (and user controllable) level.

```
<name-class-gc>  ::= GROUPING OF CLASS_NAME AS
                     SPECIFIED BY CLASS_NAME
```

The semantic checks which will be made for the name class grouping connection are as follows:

Production:

$$<\text{name-class-gc}> \longrightarrow \text{ grouping of } PC \text{ as specified by } C_1$$

Checks:

       i. Class definition for C_1 must contain "determines: C" clause

      ii. $C_2 \equiv$ STRINGS, NUMBERS, REALS, or INTEGERS

The attribute predicate used within the attribute-defined subclass definition and the subvalue derivation predicate will now be discussed. The attribute predicate is a boolean expression which relates mappings defined within the defining class C with (1) constant values, (2) other mappings within C, or (3) members of other

classes. The production sequence for the attribute predicate is as follows:

```
<attribute-predicate>  ::=  <attr-pred-term>  |
        <attribute-predicate> OR <attr-pred-term>

<attr-pred-term>  ::=  <attr-pred-factor>  |
        <attr-pred-term> AND <attr-pred-factor>

<attr-pred-factor>  ::=  <attr-pred-primary>  |
        NOT <attr-pred-primary>

<attr-pred-primary>  ::=  ( <attribute-predicate> )  |
                <simple-predicate>

<simple-predicate>  ::=  <relop-predicate>  |
                <setop-predicate>

<relop-predicate>  ::=  <mapping> <relop> <constant>  |
                <mapping> <relop> <mapping>

<setop-predicate>  ::=  <mapping> <setop> <constant>  |
                <mapping> <setop> <mapping>  |
                <mapping> <setop> CLASS_NAME

<relop>  ::=  = | ≠ | < | <= | > | >=

<setop>  ::=  IS <properly> CONTAINED IN  |
        <properly> CONTAINS

<properly>  ::=  PROPERLY | ε
```

The boolean operator precedence (NOT then AND then OR) has been built into the SDML grammar. The semantic checks which will be performed for the productions which make up the attribute predicate are as follows:

Production:

$$<relop-predicate> \longrightarrow AM_1 <relop> <constant>$$

Checks:

  i. $(AM_1)^{N_1}$ must be defined as a member attribute of C

  ii. Define C_1 as the value class of AM_1, and type($<constant>$) as the type of the nonterminal $<constant>$ (STRING_C, REAL_C, or INTEGER_C). Then:

a. if type($<$constant$>$) $=$ STRING_C then $U_{c_1} \equiv$ STRINGS

b. otherwise, $U_{c_1} \equiv$ NUMBERS, INTEGERS, or REALS

Production:

$<$relop-predicate$> \;-\!\!\to\; AM_1 \;<$relop$>\; AM_2$

Checks:

i. $(AM_i)^{N_1}$ must be defined as a member attribute of C for $i = 1,2$

ii. Given that C_i is the value class of AM_i for $i = 1,2$, then $U_{c_i} \equiv$ STRINGS, NUMBERS, INTEGERS, or REALS

iii. $U_{c_1} \equiv U_{c_2}$

Production:

$<$setop-predicate$> \;-\!\!\to\; AM_1 \;<$setop$>\; <$constant$>$

Checks:

i. $(AM_1)^{N_1}$ must be defined as a member attribute of C

ii. Since $<$constant$>$ denotes a single string or numeric constant, then $<$setop$>$ must be the "[properly] contains" set operation.

iii. Letting C_1 be the value class of AM_1, then:

a. if type($<$constant$>$) $=$ STRING_C then $U_{c_1} \equiv$ STRINGS

b. otherwise, $U_{c_1} \equiv$ NUMBERS, INTEGERS, or REALS

Production:

$<$setop-predicate$> \;-\!\!\to\; AM_1 \;<$setop$>\; AM_1$

Checks:

i. $(AM_i)^{N_1}$ must be defined as a member attribute of C for $i = 1,2$

ii. Given that C_i is the value class of AM_i for $i = 1,2$, then $U_{c_1} \equiv U_{c_2}$

Production:

$<$setop-predicate$> \;-\!\!\to\; AM_1 \;<$setop$>\; C_1$

Checks:

    i. $(AM_1)^{N_1}$ must be defined as a member attribute of C

    ii. Defining C_2 as the value class of AM_1, then $U_{C_2} \equiv U_{C_1}$

Production:

    $<\text{setop}> \;\longrightarrow\;$ is $<\text{properly}>$ contained in

Checks:

    i. if RHS of $<\text{setop}>$ is an attribute mapping, then it must be a multivalued attribute mapping

Production:

    $<\text{setop}> \;\longrightarrow\; <\text{properly}>$ contains

Checks:

    i. LHS (AM_1) must be a multivalued attribute mapping.

The format directive allows the database designer to provide a template for data to be entered for a given name class. With the format directive, comes a variety of constructs which can be used to specify the *subunits* of the domain and place restrictions on these subunits. The format directive used in the SDML grammar is a modified version of the Domain Definition Language (DDL) outlined in reference [13].

The format directive for a name class definition contains three distinct parts: (1) description clause, (2) ordering clause, and (3) violation action clause.

```
<format-directive> ::= <description-clause>
                       <ordering-clause>
                       <violation-actn-clause>
```

The description clause allows for several alternative domains for the format of the name class. Thus, each possible domain is specified in an "or"-list.

```
<description-clause>  ::=  <description-subclause>  |
        <description-clause>  OR  <description-subclause>
```

The description subclause specifies one of the possible domains for the format of the name class. The description subclause is made up of the description of the domain followed by any restrictions placed on subunits of the domain. The restrictions placed on the subunits of the domain is termed the *global where restriction.*

```
<description-subclause>  ::=  <description>  .  |
        <description>  ,  <where-restriction>  .
```

Here, additional syntax (comma's and period's) was added to the DDL grammar provided in reference [13] to resolve conflicts resulting from the DDL grammar being non-LR(1). The conflict arises because the parser cannot distinguish the "or" in the <where-restriction> from the "or" separating the <description-subclause> nonterminals in the <description-clause> production.

The description for a domain defines the subunit's which constitute the domain. The subunit's which make up the domain are separated by comma's.

```
<description>  ::=  <subunit>  |
            <description>  ,  <subunit>
```

A subunit is identified by an optional *label.* Use of the label allows this particular subunit of the domain to be used in the restriction clause as part of the domain definition. If the label is omitted, this subunit cannot be referenced further. The scope of a subunit label is only within the current domain definition.

```
<subunit>  ::=  <subunit-item>  |
        LABEL :  <subunit-item>
```

The subunit item can be defined to be any one of the following:

1. Subset of strings with an optional constraint on the strings allowed. *subunit where restriction.*

2. Subset of numbers with an optional constraint on the numbers allowed.

3. An enumeration of string constants or types. The possible types of strings are *alphabetics*, *numerics*, and *specials*.

4. An enumeration of number constants.

5. A string constant.

If only one enumeration is given, no parenthesis are required. If two or more enumerations are given, they must be enclosed in parenthesis. The restrictions placed on the subset of subunit values is termed the *subunit where restriction*. The productions which define the subunit definition follows:

```
<subunit-item>  ::= STRING <str-where-clause>  |
             NUMBER <num-where-clause>  |
             ONEOF <string-list>  |
             ONEOF <number-list>  |
             STRING_C

<str-where-clause>  ::= ε |
              WHERE <string-boolean>

<num-where-clause>  ::= ε |
              WHERE <number-boolean>

<string-list>  ::= <string-component>  |
       ( <string-list> , <string-component> )

<string-component>  ::= STRING_C |
             ALPHABETICS |
             NUMERICS |
             SPECIALS

<number-list>  ::= <number>  |
       ( <number-list> . <number> )
```

The boolean expression which restricts the subset of strings in the subunit where restriction is described next. The predicates of the boolean expression are the typical boolean operations "and", "or", and "not" as well as an "if-then-else" construct. The conditions of the boolean expression are the following:

1. A lexicographic comparison of the subunit with a string constant.

2. A scalar comparison of the size of the subunit with an integer constant ("size" operator).

3. A boolean condition which evaluates to "true" if any one of a number of given string components are found within the subunit ("has" operator).

4. A boolean user-defined function which supplies a value of "true" or "false".

The productions which make up the string subunit where restriction follows:

```
<string-boolean> ::= <string-bool-term> |
      <string-boolean> OR <string-bool-term>

<string-bool-term> ::= <string-bool-factor> |
      <string-bool-term> AND <string-bool-factor>

<string-bool-factor> ::= <string-bool-primary> |
      NOT <string-bool-primary>

<string-bool-primary> ::= ( <string-boolean> ) |
                  <string-predicate>

<string-predicate> ::= <string-condition> |
      IF <string-condition> THEN <string-predicate> FI |
      IF <string-condition> THEN <string-predicate>
                      ELSE <string-predicate> FI

<string-condition> ::= <relop> STRING_C |
                SIZE <relop> INTEGER_C |
                HAS <string-list> |
                CALL PROCEDURE_NAME
```

Notice that the precedence of the boolean operators is built in to the SDML

grammar.

The boolean expression which restricts the subset of numbers in the subunit where restriction is described next. The predicates of the boolean expression are the typical boolean operations "and", "or", and "not", an integer and real restriction, and an "if-then-else" construct, . The conditions of the boolean expression are the following:

1. A scalar comparison of the subunit with a numeric constant.

2. A boolean user-defined function which supplies a value of "true" or "false".

The productions which make up the number subunit where restriction follows:

```
<number-boolean>  ::=  <number-bool-term>  |
      <number-boolean> OR <number-bool-term>

<number-bool-term>  ::=  <number-bool-factor>  |
      <number-bool-term> AND <number-bool-factor>

<number-bool-factor>  ::=  <number-bool-primary>  |
      NOT <number-bool-primary>

<number-bool-primary>  ::=  ( <number-boolean> )  |
                  <number-predicate>

<number-predicate>  ::=  <number-condition>  |
      IF <number-condition> THEN <number-predicate> FI  |
      IF <number-condition> THEN <number-predicate>
                        ELSE <number-predicate> FI  |
      INTEGER  |
      REAL

<number-condition>  ::=  <relop>  <number>  |
              CALL PROCEDURE_NAME
```

Based on the SDML grammar definitions of the number and string subunit where restrictions, it can be seen that no additional type checking is needed for these constructs. All type checking here is forced in the SDML grammar itself.

Now for the definition of the global where restriction. The global where restriction, like the subunit where restriction, is a boolean expression which restricts the domain definition as a whole. Any labels given to subunits within the subunit definitions can be referenced within the global where restriction. The same constructs apply to the global where restriction as in the string subunit where restriction. The conditions which can be used within the global where restriction are as follows:

1. A boolean expression representing a scalar or lexicographic comparison between two subexpressions. The type of comparison depends upon the type of primitives used within the subexpressions.

2. A boolean "present" operator which returns "true" if a second substring is contained within the first.

3. A boolean user-defined function which supplies a value of "true" or "false".

The productions which begin the construction of the global where restriction are as follows:

```
<where-restriction> ::= WHERE <boolean>

<boolean> ::= <boolean-term> |
      <boolean> OR <boolean-term>

<boolean-term> ::= <boolean-factor> |
      <boolean-term> AND <boolean-factor>

<boolean-factor> ::= <boolean-primary> |
      NOT <boolean-primary>

<boolean-primary> ::= ( <boolean> ) |
                  <predicate>

<predicate> ::= <condition> |
      IF <condition-expr> THEN <predicate> FI |
      IF <condition-expr> THEN <predicate>
                      ELSE <predicate> FI

<condition> ::= <expression> <relop> <expression> |
            PRESENT ( <expression> . <expression> ) |
            CALL PROCEDURE_NAME
```

Since the expressions referenced in the global where restriction can evaluate to either strings or numerics, some type checking is required to verify the semantics. As previously defined, the type of nonterminal "expression" will be denoted as "type(<expression>)". The following type checks will be performed to the above production rules (where E_i is expression number i):

Production:

$$<condition> \longrightarrow <E_1> <relop> <E_2>$$

Checks:

    i. $\text{type}(<E_1>) = \text{type}(<E_2>)$

Production:

$$<condition> \longrightarrow \text{present} (<E_1>, <E_2>)$$

Checks:

    i. $\text{type}(<E_1>) = \text{"string"}$

    ii. $\text{type}(<E_2>) = \text{"string"}$

Some boolean operators were added to the condition of the if-then-else statements
for the global where restriction which were missing from the grammar presented in
reference [13]. The condition expression for the if-then-else can then be a boolean
expression involving the conditions of the global where restriction. The
productions for the condition expression are:

```
<condition-expr>  ::=  <condition-term>  |
       <condition-expr> AND <condition-term>

<condition-term>  ::=  <condition-factor>  |
       <condition-term> OR <condition-factor>

<condition-factor>  ::=  ( <condition-expr> )  |
              <condition>
```

In case expressions involve numeric terms, a construct for building arithmetic
expressions has been provided.

```
<expression>  ::=  <arithmetic-term>  |
       <expression> <addition-operator> <arithmetic-term>

<arithmetic-term>  ::=  <arithmetic-factor>  |
       <arithmetic-term> <multiply-operator> <arithmetic-factor>

<arithmetic-factor>  ::=  <arithmetic-primary>  |
       <arithmetic-factor> <exponent-operator> <arithmetic-primary>

<arithmetic-primary>  ::=  ( <expression> )  |
              <subexpression>
```

If an expression is a string constant, none of the production alternatives involving
the arithmetic operators should be allowed. Therefore, the following type checks
will be performed for the above productions:

Production:

$$<E> \; \rightarrow \; <E> \; <\text{add-op}> \; <T>$$

Checks:

      i. type$(<E>) = $ "numeric"

  ii. type($<T>$) = "numeric"

Production:

$$<T> \; -\!\!\rightarrow \; <T> \; <\text{mult-op}> \; <F>$$

Checks:

  i. type($<T>$) = "numeric"

  ii. type($<F>$) = "numeric"

Production:

$$<F> \; -\!\!\rightarrow \; <F> \; <\text{exp-op}> \; <P>$$

Checks:

  i. type($<F>$) = "numeric"

  ii. type($<P>$) = "numeric"

The subexpression definition supplies the following functions which can be applied to string or numeric terms:

1. Specification of an *atomic expression*, which represents an individual item. The types of atomic expressions which can be specified are:

   a. A subunit label which is defined within this domain description.

   b. A string constant.

   c. A (positive or negative) numeric constant.

   d. The star (*) operator. A star represents the current value of the domain being checked (i.e., the users input which is being checked for validity).

2. The minimum, maximum, sum, or average of a list of expressions.

3. The result of appending two string expressions.

4. The substring of a string expression between two given points.

5. The left substring of a string expression.

6. The right substring of a string expression.

7. The location of one string within another.

8. The length of a string expression.

9. Repeated applications of a subunit range to the domain definition. This alternative allows a subunit range to be repeated some number of times when considering the domain of the name class.

The productions which specify the subexpression options are:

```
<subexpression> ::= <atomic-expression> |
    <set-function> ( <expression-list> ) |
    APPEND ( <expression> , <expression> ) |
    SUBSTRING ( <expression> , <char-pos> ,
                             <char-pos> ) |
    LEFT ( <expression> , <char-pos> ) |
    RIGHT ( <expression> , <char-pos> ) |
    LOCATION ( <expression> , <expression> ) |
    LENGTH ( <expression> ) |
    REPETITIONS LABEL THROUGH LABEL

<atomic-expression> ::= LABEL |
               STRING_C |
               <number> |
               <addition-operator> <number> |
               *

<expression-list> ::= <expression> |
        <expression-list> , <expression>
```

The type checks which will be performed for the above productions are as follows:

Production:

$$<\text{subexpression}> \;\longrightarrow\; <\text{set-function}> \; (\; <E_k> \;)$$

Checks:

    i. type($<E_i>$) = "numeric" for all i in k

Production:

$$<\text{subexpression}> \;\longrightarrow\; \text{append} \; (\; <E_1>, <E_2> \;)$$

Checks:

    i. type($<E_1>$) = "string"

    ii. type($<E_2>$) = "string"

Production:

$$<\text{subexpression}> \;\longrightarrow\; \text{substring} \; (\; <E>, <CP_1>, <CP_2> \;)$$

Checks:

    i. type($<E>$) = "string"

Production:

$$<\text{subexpression}> \;\longrightarrow\; \text{left} \; (\; <E>, <CP> \;)$$

Checks:

    i. type($<E>$) = "string"

Production:

$$<\text{subexpression}> \;\longrightarrow\; \text{right} \; (\; <E>, <CP> \;)$$

Checks:

    i. type($<E>$) = "string"

Production:

$$<\text{subexpression}> \;\longrightarrow\; \text{location} \; (\; <E_1>, <E_2> \;)$$

Checks:

    i. type($<E_1>$) = "string"

ii.   type($<E_2>$) = "string"

Production:

   $<subexpression>$ $-\rightarrow$ length ($<E>$)

Checks:

   i.   type($<E>$) = "string"

The $<char\text{-}pos>$ nonterminals (and the CP_i's) describe a character position within a string. The character position could be: (1) the location of a substring within the string, (2) the location of a substring offset by an integer constant within the string, or (3) an integer constant defining a character position within the string.

```
<char-pos>  ::= STRING_C |
         STRING_C <addition-operator> INTEGER_C |
         INTEGER_C
```

The second part of the format directive is the ordering clause. This is an optional specification of how the elements of the domain will be ordered:

1. via a specified list of subunit labels,

2. no ordering will be done,

3. atomic ordering will be done; that is, lexicographic ordering will take place, or

4. via a user-defined function.

The following productions define the ordering clause for the format directive:

```
<ordering-clause>  ::=  ε  |
            ORDERING : <ordering>

<ordering>  ::=  <ordering-list>  |
         NONE  |
         ATOMIC  |
         CALL PROCEDURE_NAME

<ordering-list>  ::=  LABEL  |
            <ordering-list> , LABEL
```

Finally, a violation action can be specified, which will determine the action taken
when data is entered which does not conform to the domain specification. Either
an error can be flagged (and an optional message printed) or a user-defined function
can be called to determine the course of action. Thus, the productions which define
the violation action clause for the domain definition are as follows:

```
<violation-actn-clause>  ::=  ε  |
     VIOLATION ACTION : <violation-action>

<violation-action>  ::=  ERROR <action-message>  |
            CALL PROCEDURE_NAME

<action-message>  ::=  STRING_C  |  ε
```

The final productions in the SDML grammar define the number and constant
nonterminals used within the rest of the SDML grammar. A number is an integer
or real constant. A constant is a string constant or a number constant. A constant
also could be a CONST_MEMBER_NAME representing a string constant or a
number constant.

```
<number>  ::=  INTEGER_C  |  REAL_C

<constant>  ::=  STRING_C  |
         <number>  |
         CONST_MEMBER_NAME
```

5.4 Summary

This chapter has presented the design of the SDML grammar for the SDM model described in Chapter 2, with extensions added which were described in Chapter 3. Also presented were the semantic checks necessary which would not be detected by the simple syntax checking of the SDML parser. It was determined that some type checking within expressions was also necessary.

Chapter 5 will discuss issues involving the design of a parser for the SDML grammar. Also discussed will be the data structures used by the SDML parser to perform a complete semantic check of the SDML specification being parsed.

Chapter 5: SDML Parser Implementation

This chapter will discuss the SDML parser for the grammar presented in Chapter 4. The structure of the tables used by the parser and the structure of the parser output will also be discussed. Before getting into the design of the SDML parser, however, an introduction to parser theory will first be given.

6.1 SDML Parser Theory

To allow for ease of implementation of the SDML parser, the Yacc *(Yet Another Compiler-Compiler)* program will be used to automatically generate a parser with the desired functionality. Yacc accepts a general class of LR(1) grammars which may or may not contain ambiguities. Yacc will resolve these ambiguities with *disambiguating rules*. Because of these disambiguating rules and the parsing algorithm which Yacc uses, SDML grammar ambiguity is not a concern once the grammar is accepted by Yacc with no shift-reduce or reduce-reduce conflicts reported.

Yacc generates a *shift-reduce* parser for the supplied grammar. The following sections will describe shift-reduce parsing theory and error recovery used by the Yacc program.

6.1.1 *Shift-Reduce Parser Theory* A shift-reduce parser is a bottom-up LR parsing technique which reduces an input token stream left-to-right producing a rightmost derivation in reverse. A *rightmost derivation* is achieved by replacing the rightmost nonterminal at every derivation step. A rightmost derivation is also referred to as a *canonical derivation*.

Using the *rm* subscript on the derivation symbol $(=>)$ to denote a rightmost derivation, and given a rightmost derivation sequence

$$S_s \ = >^*_{rm} \ \alpha.$$

then α is a *right-sentential form* for the grammar Γ.

The *handle* of a right-sentential form $\alpha \beta w$ of Γ is a substring β of $\alpha \beta w$ such that

$$S_s \ = >^*_{rm} \ \alpha A w \ => \ \alpha \beta w \text{ and } A \ \rightarrow \ \beta.$$

where $\alpha A w$ is the previous right-sentential form and $w \in \Sigma^*$. A rightmost derivation in reverse (also called a *canonical reduction sequence*) is obtained by *handle pruning.* Handle pruning is the act of building and reducing handles from the input token stream. In a canonical reduction sequence, the leftmost handle is located and reduced to form the previous right-sentential form. This single reduction is called a *canonical reduction* and is performed as follows:

i. Define w as the input token stream which is to be parsed. Thus $\nu_n = w$ represents the nth right-sentential form of the desired canonical derivation. The desired canonical derivation would be as follows:

$$S_s = \nu_0 \ = >_{rm} \ \nu_1 \ = >_{rm} \ \cdots \ = >_{rm} \ \nu_{n-1} \ = >_{rm} \ \nu_n \ = \ w.$$

ii. Locate the handle β_n in ν_n and replace β_n by the LHS of some production $A_n \ \rightarrow \ \beta_n$ to obtain the $(n-1)^{st}$ right-sentential form ν_{n-1}.

iii. Repeat step (ii.) until (assuming a successful parse) the 0th right-sentential form (i.e., $\nu_0 = S_r$) is found. When the last handle can be replaced with the start symbol, a successful parse is accomplished.

Parsers which perform this type of bottom-up parsing by handle pruning are

called shift-reduce parsers because tokens are *shifted* onto the token stack until a handle is found, when the handle is *reduced* to a nonterminal symbol representing the LHS of the production corresponding to the handle on the RHS.

The Yacc program performs such a shift-reduce parsing algorithm. Now that the basic theory behind shift-reduce LR parser algorithms is understood, the conflict and error recovery capabilities of Yacc will be discussed.

6.1.2 *Yacc Grammar Conflicts and Error Recovery* Based on the structure of the grammar supplied to Yacc, two types of conflicts may arise: (1) shift-reduce conflicts and (2) reduce-reduce conflicts.

Shift-reduce conflicts arise when the parser does not know whether to shift the current input token onto the stack as part of a handle or to reduce the current handle on the stack using one of the grammar rules. Shift-reduce conflicts typically result when an ambiguous grammar is specified.

Reduce-reduce conflicts arise when the parser does not have enough information to select one of two grammar rules when a handle can be reduced, even with the current look-ahead token. Reduce-reduce conflicts result when a non-LR(1) grammar is supplied to Yacc.

The SDML grammar listed in Appendix 4 contains no shift-reduce or reduce-reduce conflicts.

Yacc provides an error recovery mechanism in case of an error in parsing the input token stream. This mechanism uses the technique of discarding input tokens until a *safe-point* is reached on the stack, and attempting to continue parsing from that point. Yacc allows the grammar to include a special terminal symbol, *error*, which

defines a safe-point for the grammar (or a point where errors will most likely occur). When the Yacc parser detects an error in the input stream, it pops its stack until a safe-point marker is reached and attempts to continue from that point. Yacc will remain in an error state until three tokens have been successfully read and shifted onto the rebuilt stack.

6.2 The SDML Parser

The SDML parser will be automatically generated by the Yacc [15] program. Within the Yacc grammar specification, blocks of code called *actions* are supplied with certain grammar rules to specify some action to take when the given handle is found and reduced using that particular grammar rule.

The Lex [14] program will automatically generate the lexical analyzer which will tokenize the input stream representing the SDML specification. Lex will pass the next token from the input stream to Yacc, which will be parsing the tokens received. Additional implementation required (besides supplying the tokens and grammar to Lex and Yacc respectively) will be to supply the blocks of code within the Lex and Yacc input files and the subroutines used by these blocks of code to perform the actions necessary by the SDML parser.

The SDML parser will be a two-pass parser. Pass 1 of the parser will check syntax (by performing an LR(1) parse of the input token stream) and load symbols and constants into the symbol table. Only defined (as opposed to referenced) class names, attribute names, label names, and constant member names will be loaded into the symbol table during pass 1. The reason for this is two-fold:

1. attribute names must be entered in the symbol table with the underlying

base class as part of the key. Therefore, if an attribute name is referenced before it is defined, the reference will not know the class to which the symbol belongs. An attributes class definition when referenced depends on the context in which the attribute appears. This attribute definition check will be done by the parser during pass 2 of the compiler as part on the semantic checks for the reference.

2. undefined symbol references during pass 2 are recognized simply by their non-existence in the symbol table.

To make the code consistent for all symbol type definitions, all symbols will be handled in this fashion. Constant symbols, however, have no basis for definition and have no uniqueness criteria and will, therefore, be loaded into the symbol table when recognized on pass 1 of the SDML parser.

The following sections describe the various aspects of the SDML parser including symbol table administration, semantic checks, and parser output.

6.3 Restrictions

The major restriction enforced by the SDML parser is when naming class, attribute, label, procedure, and constant member names:

- Class names must begin with an upper-case character and can be followed by zero or more upper-case characters, underscores, or digits.

- Attribute names must begin with an upper-case character and can be followed by one or more lower-case characters, underscores, or digits.

- Label names must begin with a lower-case character and can be followed by zero or more lower-case characters, underscores, or digits.

- Procedure names must begin with an underscore and can be followed by one or more lower-case characters, underscores, or digits.

- Constant member names must begin with an underscore and can be followed by one or more upper-case characters, underscores, or digits.

The other restriction enforced by the SDML parser is that all eventual parent classes must be defined prior to definition of a member attribute for a nonbase class. The reason for this restriction is so the uniqueness condition can be checked for attribute definition. This implies that the underlying base class for the class being defined is known so that it can be entered into the symbol table with the attribute name.

6.3.1 *The Symbol Table* The symbol table for the SDML parser will hold all label and constant symbols recognized by the lexical analyzer. Each type of symbol which can be loaded into the symbol table must be unique with respect to some defined scope. Figure 6-1 lists each symbol which can be loaded into the symbol table and its scope of uniqueness (w.r.t. means *with respect to*).

Symbol Type	Uniqueness
Class Name	w.r.t. SDM schema
Attribute Name	w.r.t. Underlying base class
Label Name	w.r.t. Description subclause
Procedure Name	w.r.t. SDM schema
Constant Member Name	w.r.t. SDM schema
String Constant	None
Integer Constant	None
Real Constant	None

Figure 6-1. SDML Symbol Types and Uniqueness

To implement the different symbol uniqueness scopes, the symbol table *key* will consist of three elements: (1) symbol type, (2) supplementary key data (providing uniqueness), and (3) symbol name. Figure 6-2 shows the layout of the symbol table key with the size (in bytes) of the item in parenthesis.

```
| type (2) |  sup_data (4)  | symbol name (80)          |
```

Figure 6-2. Symbol Table Key Layout

The symbol table is indexed by a hash table which is kept within the *hsearch* *UNIX*[TM] system library function. Hsearch provides a Knuth Algorithm D hashing function from the key. An hsearch hash table entry consists of two pointers: (1) a key pointer and (2) a supplementary data pointer. The key pointer simply gives the location within the application program (here the SDML parser) where the key resides for that entry and the supplementary data pointer gives the location within the application program where any additional data is stored. The size of the hash table is determined at the time that the first entry must be made. If the user did not change the minimum size of the hash table (using the "%h" control), a default minimum size is used.

The key and the additional data for a symbol table entry are allocated within the

SDML parser. The key for the symbol table entry is named *st_key* and the additional data structure for the symbol table entry is named *st_data*. The additional data for the symbol table entry contain the following data:

name   : pointer to name of symbol within the key

type   : type of symbol (part of key)

sup_data : supplementary key data (part of key)

usage   : symbol usage count

table_p : pointer to table where symbol specific data is stored. This can be either a class structure or an attribute structure.

The size of the symbol table is defined the same as the size of the hash table. Attribute structure location is exactly the same as class structure location. The only difference is the template used for the memory pointed to by the *table_p* pointer.

Additional synonymous names for classes and attributes are also entered into the symbol table but have a *table_p* which points to the primary class or attribute structure. Thus, any references to synonymous names for classes or attributes will ultimately result in the same class or attribute structure being referenced.

6.3.2 *Class and Attribute Structures* The class and attribute structures contain all information pertinent to the class and attribute definitions. These structures contain the data which defines how the class or attribute was defined by the user in the SDM schema. The class structure contains the following information:

- index of symbol table entry identifying class

- base vs. nonbase indication

- parent class symbol table index

- underlying base class symbol table index

- description text (if any)

- if base class, whether duplicates are allowed or not

- number of member attributes defined in class

- index of first member attribute defined in class

- number of class attributes defined in class

- index of first class attribute defined in class

- if base class, identifiers which uniquely identify a member of this class

- if nonbase class, interclass connection defining how this class is formed from its parent class

- if name nonbase class, the class symbol table index of class which has a value class determined by this class

- if name nonbase class, the number of constant members which are defined within this class

- if name nonbase class, the first constant member symbol table index

The number of classes and attributes which can be defined is a user-controllable value which can be changed with the "%c" and "%a" control respectively. The attribute structure contains the following data:

- index of symbol table entry for class defining this attribute

- index of next attribute in list (or nil)

- member vs. class attribute

- description text (if any)

- index of symbol table entry for value class

- index of symbol table entry for inverse attribute

- indication of match relationship. If match exists then contains: (1) derived attribute index, (2) matching class index, and (3) condition attribute index.

- indication of derivation and actual derivation

- single valued vs. multivalued

- if multivalued, then size range

- may not be null

- not changeable

- exhausts value class

- no overlap in values

- attribute assertion (if any)

Based on the contents of the class and attribute structures given above, member and class attributes are lined using a single link-list from the defining class with the attribute link of the last class definition being nil. Each attribute has a index pointer back to the defining class.

6.3.3 *Semantic Checks* The SDML grammar specification supplied to Yacc will contain all necessary semantic checks identified in Chapter 4 as actions for the appropriate grammar rule. All semantic checks will be performed in pass 2 of the SDML parser after all symbol table entries are built. Any semantic errors detected will not halt pass 2 of the compile but will simply issue warning messages with a line number. This will allow all semantic verifications to be made at once; thus allowing the user to correct semantic errors at one time.

6.3.4 *SDML Parser Output* The output of the SDML parser will be the structures which are built during passes 1 and 2 of the compiler. These structures will represent the class and attribute definitions which were specified in the SDML specification supplied by the user. The hash table, however, will not be generated

as output since it's purpose was realized during pass 2 of the parser for symbol lookup. For this reason, all references to classes or attributes are via a direct index into the symbol table. The structure dump is requested by specifying $-d$ on the command line (indicating that data dictionary generation is to be performed).

The generated SDM schema can be directed to an output file by specifying the $-o$ option on the command line. An output of the symbol table can be requested by using the $-s$ option on the command line Finally, a verbose statistics report can be requested by requesting the $-v$ option on the command line.

The manual page associated with the SDML parser is shown in Appendix 6.

6.4 Summary

This chapter has discussed the design of the SDML parser for the grammar designed in Chapter 4. The design of the SDML parser is primarily that of specifying a suitable grammar to the Yacc process and supplying actions to be performed when certain reductions are performed. The SDML grammar was designed to be easily expandable and maintainable. This ease of expansion and maintainability is a benefit which is inherent in the use of Yacc.

Chapter 6: Conclusion

This paper has developed a grammar defining a Semantic Database Model (SDM) language and discussed the design of a parser to parse the language. The Semantic Database Model Language (SDML) is used as an intermediate in the generation of a static data dictionary.

Some extensions to the SDM model which would both enhance capabilities provided by the SDM model and enhance the semantic integrity provided by the SDML specification over the SDM model which were introduced follows:

- assertions, which allow the database designer to include statements of fact within the SDML specification, thus greatly improving the semantic integrity of the SDML specification over the SDM model,

- grouping name classes, which allow the value of one attribute to determine the value class of another attribute,

- constant name class members, which allow the use of a constant member of a name class within expressions in the SDML specification, and

- sizeof class and attribute operator, which allows the reference to the number of members in a class or multivalued attribute.

An SDML grammar which includes the extensions was created. Along with the presentation of the production rules for the grammar, the semantic checks which should be made are listed for each production. These semantic checks will provide a mechanism for reporting misuse of SDM features. The semantic checks ensure

that the value class of a member attribute is consistent with the derivation for that attribute and that the interclass connection for a nonbase class is consistent with the underlying base class and all classes referenced in the connection. The semantic checks also verify that the member derivation rules which protect against inconsistent attribute derivations are enforced.

The design of the SDML parser for the grammar was developed. The Yacc program was used to automatically generate a shift-reduce parser for the SDML grammar. Symbol table administration and semantic check routines were added as actions to SDML grammar rules to complete the SDML parser design.

7.1 Future Research Areas

Chapter 3 introduced the concept of assertions into the SDML specification. The assertions provided are conceived as a minimal set of assertions. Therefore, any additional assertions which could be identified would add even more to the semantic integrity of the SDML specification.

Part of the failure action for an assertion and the violation action for format directives is the ability for the database designer to specify user-defined procedures within the SDML specification to generalize the failure action. The SDML specification could be expanded to allow definition of these procedures (procedure specifications) within the SDML specification.

Chapter 3 also talked about the possibility of specifying an additional failure action which would indicate the manner in which the database should be corrected in order to return to a valid database state. This idea is implicit in the SDML specification presented here in that database repair is considered a function of a

user-defined procedure. This concept, however, could be explicitly specified in the SDML specification, thus providing more semantic information within the model.

Some of the constructs provided by the SDM model presented by Hammer and McLeod can be difficult to understand. For this reason, the SDM model will probably not be realized as a widely used database modeling mechanism. Some of the constructs provided in the SDM model could be combined to form a more uniform set of capabilities instead of seamingly ad-hoc capabilities. On a larger scale, the development of a new SDM model, based on concepts from the old SDM, may be desirable. The new SDM model should provide a cohesive set of capabilities (probably not unlike the old SDM) which are easy to understand and use.

References

1. Hammer, M. and McLeod, D., "Database Description with SDM: A Semantic Database Model," ACM T-DS, Vol. 6, No. 3, Sept. 1981, pp. 351-386.

2. Hammer, M. and McLeod, D., "The Semantic Data Model: A Modelling Mechanism for Data Base Applications," ACP SIGMOD, Auston, Texas, 1978, pp. 26-36.

3. Hammer, M. and McLeod, D., "A Framework for Data Base Semantic Integrity," Proc. 2nd Int. Conf. Software Engineering, San Francisco, Calif., Oct. 1976, pp. 498-504.

4. Hammer, M. and McLeod, D., "Semantic Integrity In a Relational Data Base System," Proc. Int. Conf. Very Large Databases, Framingham, Mass., Sept. 1975, pp. 25-47.

5. McLeod, D. and King, R., "Applying a Semantic Database Model," in Entity-Relationship Approach to Systems Analysis and Design, P. P. Chen [Ed.], North-Holland, 1980, pp. 193-210.

6. Roussopoulos, N. and Mylopoulos, J., "Using Semantic Networks for Data Base Management," Proc. Int Conf. Very Large Databases, Framingham, Mass., Sept. 1975, pp. 144-172.

7. Su, S. Y. W. and Lo, D. H., "A Semantic Association Model for Conceptual Database Design," in Entity-Relationship Approach to System Analysis and Design, P. P. Chen [Ed.], North-Holland, 1980, pp. 169-192.

8. Roussopoulos, N., "ADD: Algebraic Data Definition," Proc. 6th Texas Conf. Computing Systems, Auston, Texas, Nov. 1977, pp. 1C-16 to 1C-25.

9. Lee, R. M. and Gerritsen, R., "Extended Semantics for Generalization Hierarchies," ACM SIGMOD, Auston, Texas, 1978, pp. 18-25.

10. Barrett, W. A. and Couch, J. D. [1979], *Compiler Construction: Theory and Practice*, Science Research Associates.

11. Gries, D. [1971], *Compiler Construction for Digital Computers*, Wiley, New York.

12. Aho, A. V. and Ullman, J. D. [1979], *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts.

13. McLeod, D. J., "High Level Definition of Abstract Domains in a Relational Data Base System," Computer Languages, Vol. 2, No. 3, 1977, pp. 61-73.

14. Lesk, M. E. and Schmidt, E., "Lex - A Lexical Analyzer Generator," Bell Laboratories, Murray Hill, New Jersey.

15. Johnson, S. C., "Yacc: Yet Another Compiler-Compiler," Bell Laboratories, Murray Hill, New Jersey.

Appendix 1: Example SDM Schema

This Appendix contains an example of an SDM schema to aid in the understanding of the concepts presented in Chapter 2 of this paper. This example represents the SDM schema, as it was presented by Hammer and McLeod in reference [1].

CARS and AUTOMOBILES
   description: All cars produced by a plant.
           This is a base class definition.   .
   duplicates not allowed
   member attributes:

      Make
        value class: CAR_TYPES
        may not be null
        not changeable

      Model
        value class: CAR_MODELS
        may not be null
        not changeable

      Year
        value class: YEARS
        may not be null
        not changeable

      Exterior_color
        value class: COLORS

      Interior_color
        value class: COLORS

      Base_price
        value class: PRICES

      Sticker_price
        value class: PRICES
        derivation: = Base_price + sum(Options.Price)

      Options
        description: Options on car
        value class: OPTIONS

Dealership
   description: Dealership stocking this car.
               This attribute shows the inverse
               relationship. Here, Dealership is
               the DEALER with this car in stock.
   value class: DEALERS   .
   inverse: Cars_in_stock

Vin
   description: Vehicle Identification Number
   value class: VIN_CODES
   may not be null
   not changeable

Owner
   description: Ultimate owner of vehicle until
               purchased by a consumer. This attribute
               shows the matching relationship. Here,
               Owner is the owner of the Dealership
               which has this car in stock.
   value class: OWNERS
   match: Owned_by of DEALERS on Cars_in_stock

identifiers:
   Vin

DEALERS, DEALERSHIPS
   description: Car dealerships which stock the cars built
             by a plant. This is a base class definition.
   member attributes:

Name
   description: Dealership name
   value class: DEALER_NAMES

Address
   value class: ADDRESS

Dealer_code
   description: Code uniquely identifying a dealership
   value class: DEALER_CODES
   may not be null

Makes
   description: Makes of cars which this dealer carries
   value class: CAR_TYPES
   multivalued

Cars_in_stock
    description: Cars which this dealership has in stock.
                This attribute closes the symmetrical
                inverse relationship with the Dealership
                attribute of class CARS.
    value class: CARS
    inverse: Dealership
    multivalued

Owned_by
    description: Owners of this dealership
    value class: OWNERS
    inverse: Dealers_owned

Employees
    description: Employees working as sales persons for
                this dealership.
    value class: PERSON_NAMES
    multivalued with size between 1 and 20

class attributes:

Number_dealer_cars
    description: Number of cars in dealership.
    value class: INTEGERS
    derivation: number of members in Cars_in_stock

identifiers:
    Dealer_code

OWNERS
    description: This class contains all owners of car
                dealerships. This is a base class definition.
    member attributes:

Name
    value class: PERSON_NAMES

Address
    value class: ADDRESS

Dealers_owned
    description: This attribute is the dealerships
                which this person or persons own.
    value class: DEALERS
    inverse: Owned_by
    multivalued

identifiers:
    Name

OPTIONS
   description: This base class contains all available
               car options.
   member attributes:

      Type
         description: Type of option
         value class: OPTION_TYPES

      Price
         description: Price of options
         value class: PRICES

   identifiers:
      Type

CARS_SOLD
   description: This class contains all cars sold to a
               customer.
   interclass connection: subclass of CARS where specified
   member attributes:

      Sold_to
         description: Customer buying the car.
         value class: PERSON_NAMES
         may not be null

      Sold_by
         description: Salesman selling car to customer.
         value class: PERSON_NAMES
         may not be null

      Customer_address
         value class: ADDRESS

      Date_sold
         value class: DATES

      Selling_price
         description: Final selling price of car.
         value class: PRICES

   class attributes:

      Number_of_cars_sold
         value class: INTEGERS
         derivation: number of unique members in this class

SCHEDULED_PREPS
description: This class contains all scheduled car
preparations. This base class definition
is an example of an event class definition.
interclass connection: subclass of CARS where specified
member attributes:

Date_of_preparation
value class: DATES

class attributes:

Number_of_sched_preps
description: number of cars to be prepared
value class: INTEGERS
derivation: number of unique members in this class

BUICKS
description: This class contains all Buick cars. This
nonbase class shows the use of the
attribute-defined subclass connection utilizing
the simple attribute predicate.
interclass connection: subclass of CARS where Make = 'Buick'
member attributes:

Model
description: this shows the concept of restricting
the value class of an inherited member
attribute.
value class: BUICK_MODELS

SOMERSETS
description: This class contains all Buick Somerset model
cars. This nonbase class definition shows
the use of the attribute-defined subclass
connection utilizing the compound attribute
predicate. Note that the interclass connection
for this class could also have been "subclass
of BUICKS where Model = 'Somerset'".
interclass connection: subclass of CARS where
Make = 'Buick' and
Model = 'Somerset'

PREPARED_CARS
description: This class contains all prepared cars for
any dealership. This nonbase class definition
shows the use of the user-controllable
subclass connection.
interclass connection: subclass of CARS where specified

PREPARED_BUICKS
> description: This class contains all prepared Buick's for
> any dealership. This nonbase class definition
> shows the use of the intersection set-operator-defined
> subclass connection. Note that the interclass
> connection for this class could also have been
> "subclass of PREPARED_CARS where Make = 'Buick'".
>
> interclass connection: subclass of CARS where is in BUICKS and
> is in PREPARED_CARS

BUICK_DEALERS
> description: This class contains all of the dealers which
> sell Buick's. This nonbase class definition shows
> the use of the existence subclass connection.
> Note that the interclass connection for this
> class could also have been "subclass of DEALERS
> where Makes contains 'Buick'" (which is an
> attribute-defined subclass connection).
>
> interclass connection: subclass of DEALERS where is a value of
> Dealership of BUICKS

CAR_MODEL_GROUPS
> description: This class groups cars into models. This
> nonbase class definition shows the use of the
> expression-defined grouping class. Each member
> of this nonbase class is a class containing all
> models for cars. Note that the class definition
> also indicates that a SDM class was explicitly
> defined for the Buick Somerset make and model.
>
> interclass connection: grouping of BUICKS on common value of
> Make and Model
>
> groups defined as classes are SOMERSETS

DEALER_PREP_CARS
> description: This class groups prepared cars for specific
> dealerships. This nonbase class definition
> also shows the use of the expression-defined
> grouping class.
>
> interclass connection: grouping of PREPARED_CARS on common value
> of Dealership

CAR_TYPES
> description: This is the list of all available car types.
> interclass connection: subclass of STRINGS where specified

CAR_MODELS
> description: This is the list of all available car models.
> interclass connection: subclass of STRINGS where specified

BUICK_MODELS
   description: This is the list of all possible buick car models.
   interclass connection: subclass of STRINGS where specified

YEARS
   description: This is the format of a year.
   interclass connection: subclass of STRINGS where format is
                   number where integer and $>=$ 70 and $<=$ 99

DATES
   description: Calendar dates in the range "1/1/70" to "12/31/99".
   interclass connection: subclass of STRINGS where format is
                   month: number where integer and
                           $>=1$ and $<=12$
                   '/'
                   day: number where integer and
                        $>=1$ and $<=31$
                   '/'
                   year: number where integer and
                         $>=70$ and $<=99$
                   where (if (month=4 or month=5 or
                             month=9 or month=11) then
                             day $<=30$) and
                         (if month=2 then day $<=29$)
                   ordering: year, month, day

COLORS
   description: This is the available interior and exterior
               car colors.
   interclass connection: subclass of STRINGS where specified

PRICES
   description: This is the possible car prices.
   interclass connection: subclass of STRINGS where format is
                   number where $>=$ 0
                   where length(right(*, '.'+1)) $=$ 2
                         or not present(*, '.')
                   ordering: value

VIN_CODES
   description: This is the format of the Vehicle Identification
                  Number Codes. For example: 1G4NK27U1GC612345.
   interclass connection: subclass of STRINGS where format is
                  '1G'
                  number where integer and $>=0$ and $<=9$
                  string where has alphabetics and size = 2
                  string where has numerics and size = 2
            engine_code: oneof 'U', 'G'
                  number where integer and $>=0$ and $<=9$
            model_year: string where has alphabetics and size = 1
            assy_plant: string where has alphabetics and size = 1
                  string where has numerics and size = 6
            ordering: call _vin_ordering

DEALER_NAMES
   description: Dealership names
   interclass connection: subclass of STRINGS

ADDRESS
   description: Street Address
   interclass connection: subclass of STRINGS

DEALER_CODES
   description: Dealership identifier codes
   interclass connection: subclass of STRINGS where format is
                  number where integer

PERSON_NAMES
   description: Personal Names
   interclass connection: subclass of STRINGS

OPTION_TYPES
   description: Option types
   interclass connection: subclass of STRINGS where specified

Appendix 2: SDM Structure Specification

The structure specification used here and in Appendix 3 utilize the following notational conveniences:

1. Productions are shown in the form $LHS \leftarrow RHS$, which is read: LHS "is made up of" RHS.

2. The RHS of each production is indented once to enhance reading ability. All further indentation is typically supplied (but not required) within the SDM schema.

3. X^+ means one or more occurrences of X, concatenated together.

4. $<X>$ means that one or more occurrences of X with appropriate separators (e.g., comma's or and's) are allowed.

5. $<<X>>$ means that one or more occurrences of X are listed vertically with no separators.

6. $\{ X \mid Y \}$ means that exactly one of X or Y can be used.

7. $[X]$ means that X is optional.

8. A meta-description will be enclosed in stars (e.g., * this is a meta-description *).

9. Reducible constructs are shown in upper-case, whereas all non-reducible constructs are shown in lower-case.

Those productions or lines marked with an asterik (*) are extensions to the SDM schema presented in Chapter 2.

```
SCHEMA ←
     <<CLASS>>

CLASS ←
     { BASE_CLASS | NONBASE_CLASS | NAME_CLASS }

BASE_CLASS ←
     <CLASS_NAME>
        [description: DESCRIPTION_TEXT]
        [BASE_CLASS_FEATURES]
        member attributes:
           <<MEMBER_ATTRIBUTE>>
        [class attributes:
           <<CLASS_ATTRIBUTE>>]
        identifiers:
           { <IDENTIFIER> | none }

NONBASE_CLASS ←
     <CLASS_NAME>
        [description: DESCRIPTION_TEXT]
        interclass connection: INTERCLASS_CONNECTION
        [member attributes:
           <<MEMBER_ATTRIBUTE>>]
        [class attributes:
           <<CLASS_ATTRIBUTE>>]

NAME_CLASS ←
     <CLASS_NAME>
        [description: DESCRIPTION_TEXT]
        interclass_connection: NAME_CLASS_IC
*       [determines: CLASS_NAME]
*       [constant members: CONSTANT_MEMBER_NAME]

BASE_CLASS_FEATURES ←
     { duplicates allowed | duplicates not allowed }

MEMBER_ATTRIBUTE ←
     <ATTRIBUTE_NAME>
        [description: DESCRIPTION_TEXT]
        value class: CLASS_NAME
        [inverse: ATTRIBUTE_NAME]
        [{ match: ATTRIBUTE_NAME of CLASS_NAME on ATTRIBUTE_NAME |
          derivation: MEMBER_ATTRIBUTE_DERIVATION }]
        [{ single valued |
          multivalued [with size between INTEGER and INTEGER] }]
        [may not be null]
        [not changeable]
        [exhausts value class]
        [no overlap in values]
*       [<ATTRIBUTE_ASSERTION>]
```

```
CLASS_ATTRIBUTE  ←
      <ATTRIBUTE_NAME>
        [description: DESCRIPTION_TEXT]
        value class: CLASS_NAME
        [derivation: CLASS_ATTRIBUTE_DERIVATION]
        [{ single valued |
          multivalued [with size between INTEGER and INTEGER] }]
        [may not be null]
        [not changeable]
*       [<ATTRIBUTE_ASSERTION>]

* ATTRIBUTE_ASSERTION  ←
        assertion: { ASSERTION_EXPRESSION | call PROCEDURE_NAME }
      [failure action: FAILURE_ACTION]

* ASSERTION_EXPRESSION  ←
        MAPPING_EXPRESSION = MAPPING_EXPRESSION

* FAILURE_ACTION  ←
        { call PROCEDURE_NAME |
          error [STRING] |
          warning STRING }

IDENTIFIER  ←
        { ATTRIBUTE_NAME |
          IDENTIFIER + ATTRIBUTE_NAME }

MEMBER_ATTRIBUTE_DERIVATION  ←
        { INTERATTRIBUTE_DERIVATION |
          MEMBER-SPECIFIC_DERIVATION}

CLASS_ATTRIBUTE_DERIVATION  ←
        { INTERATTRIBUTE_DERIVATION |
          CLASS-SPECIFIC_DERIVATION}

INTERATTRIBUTE_DERIVATION  ←
        { same as ATTRIBUTE_NAME |
          subvalue of MAPPING where {is in CLASS_NAME |
                                ATTRIBUTE_PREDICATE } |
          where { is in MAPPING and is in MAPPING |
                is in MAPPING or is in MAPPING |
                is in MAPPING and is not in MAPPING} |
          = MAPPING_EXPRESSION |
          number of [unique] members in MAPPING }

MEMBER-SPECIFIC_DERIVATION  ←
        { order by [{ increasing | decreasing }] <MAPPING>
                [within <MAPPING>] |
          if in CLASS_NAME |
          { up to INTEGER | all } levels of values of
                                ATTRIBUTE_NAME }
```

```
CLASS-SPECIFIC_DERIVATION ←
      { number of [unique] members in this class |
        SET_FUNCTION of ATTRIBUTE_NAME over members of this class }

MAPPING_EXPRESSION ←
      { MAPPING |
        ( MAPPING_EXPRESSION ) |
        MAPPING_EXPRESSION NUMBER_OPERATOR MAPPING_EXPRESSION |
        SET_FUNCTION ( MAPPING ) |
*      sizeof( MAPPING ) |
*      sizeof( CLASS_NAME ) }

MAPPING ←
      { ATTRIBUTE_NAME |
        MAPPING . ATTRIBUTE_NAME }

NUMBER_OPERATOR ←
      { + | - | * | / | ! }

SET_FUNCTION ←
      { minimum | maximum | average | sum }

INTERCLASS_CONNECTION ←
      { SUBCLASS_CONNECTION |
        GROUPING_CONNECTION }
*      [<INTERCLASS_ASSERTION>]

* INTERCLASS_ASSERTION ←
        interclass assertion: INTERCLASS_CONNECTION
        [FAILURE_ACTION]

NAME_CLASS_IC ←
      { NAME_CLASS_SC | NAME_CLASS_GC }

NAME_CLASS_SC ←
        subclass of STRINGS [where NAME_CLASS_SCPRED]

NAME_CLASS_SCPRED ←
      { specified |
        format is FORMAT_DIRECTIVE /*seeAppendix 3* /}

* NAME_CLASS_GC ←
        grouping of STRINGS where specified by CLASS_NAME

SUBCLASS_CONNECTION ←
        subclass of CLASS_NAME where SUBCLASS_PREDICATE
```

GROUPING_CONNECTION ←
    { grouping of CLASS_NAME on common value of <ATTRIBUTE_NAME>
      [groups defined as classes are <CLASS_NAME>] |
    grouping of CLASS_NAME consisting of classes <CLASS_NAME> |
    grouping of CLASS_NAME as specified }

SUBCLASS_PREDICATE ←
    { ATTRIBUTE_PREDICATE |
     specified |
     { is in CLASS_NAME and is in CLASS_NAME |
      is not in CLASS_NAME |
      is in CLASS_NAME or is in CLASS_NAME } |
     is a value of ATTRIBUTE_NAME of CLASS_NAME }

ATTRIBUTE_PREDICATE ←
    { SIMPLE_PREDICATE |
    (ATTRIBUTE_PREDICATE) |
    not ATTRIBUTE_PREDICATE |
    ATTRIBUTE_PREDICATE and ATTRIBUTE_PREDICATE |
    ATTRIBUTE_PREDICATE or ATTRIBUTE_PREDICATE }

SIMPLE_PREDICATE ←
    { MAPPING RELOP { CONSTANT | MAPPING } |
    MAPPING SETOP { CONSTANT | MAPPING | CLASS_NAME } }

RELOP ←
    { = | ¬= | < | <= | > | >= }

SETOP ←
    { is [properly] contained in |
    [properly] contains }

CLASS_NAME ←
    UPPER_CASE *NOT_LOWER_CASE*+

ATTRIBUTE_NAME ←
    UPPER_CASE *NOT_UPPER_CASE*+

PROCEDURE_NAME ←
    "_" *NOT_UPPER_CASE*+

CONSTANT_MEMBER_NAME ←
    "_" *NOT_LOWER_CASE*+

DESCRIPTION_TEXT ←
    "{" * any character string except } character * "}"

INTEGER ←
    DIGIT+

```
REAL  ←
      INTEGER . INTEGER

NUMBER  ←
      { INTEGER | REAL }

CONSTANT  ←
      { NUMBER | STRING }

UPPER_CASE  ←
      { A | B | ... | Z }

LOWER_CASE  ←
      { a | b | ... | z }

DIGIT  ←
      { 0 | 1 | ... | 9 }

NOT_LOWER_CASE  ←
      { UPPER_CASE | _ | DIGIT }

NOT_UPPER_CASE  ←
      { LOWER_CASE | _ | DIGIT }

STRING  ←
      "'" * any printable character except ' character * "'"
```

Appendix 3: Domain Definition Language (DDL) Structure

See Appendix 2 for a description of the notation used in the following structure specification.

```
FORMAT_DIRECTIVE ←
      DESCRIPTION_CLAUSE
      [ORDERING_CLAUSE]
      [VIOLATION_ACTION_CLAUSE]

DESCRIPTION_CLAUSE ≤
      { DESCRIPTION_SUBCLAUSE |
        DESCRIPTION_CLAUSE or DESCRIPTION_SUBCLAUSE }

DESCRIPTION_SUBCLAUSE ←
      DESCRIPTION
      [, WHERE-RESTRICTION] .

DESCRIPTION ←
      <SUBUNIT>

SUBUNIT ←
      [LABEL :] SUBUNIT_ITEM

SUBUNIT_ITEM ←
      { string [where STRING_BOOLEAN] |
        number [where NUMBER_BOOLEAN] |
        oneof STRING_LIST |
        oneof NUMBER_LIST |
        STRING }

STRING_LIST ←
      { STRING_COMPONENT |
        ( <STRING_COMPONENT> ) }

NUMBER_LIST ←
      { NUMBER |
        ( <NUMBER> ) }

STRING_COMPONENT ←
      { STRING | alphabetics | numerics | specials }

STRING_BOOLEAN ←
      { STRING_PREDICATE |
        ( STRING_BOOLEAN ) |
        not STRING_BOOLEAN |
        STRING_BOOLEAN and STRING_BOOLEAN |
        STRING_BOOLEAN or STRING_BOOLEAN }
```

STRING_PREDICATE ←
 { if STRING_CONDITION then STRING_PREDICATE |
      [else STRING_PREDICATE] fi |
 STRING_CONDITION }

STRING_CONDITION ←
 { RELOP STRING |
 size RELOP INTEGER |
 has STRING_LIST |
 call PROCEDURE_NAME }

NUMBER_BOOLEAN ←
 { NUMBER_PREDICATE |
 (NUMBER_BOOLEAN) |
 not NUMBER_BOOLEAN |
 NUMBER_BOOLEAN and NUMBER_BOOLEAN |
 NUMBER_BOOLEAN or NUMBER_BOOLEAN }

NUMBER_PREDICATE ←
 { if NUMBER_CONDITION then NUMBER_PREDICATE |
      [else NUMBER_PREDICATE] fi |
 integer |
 real |
 NUMBER_CONDITION }

NUMBER_CONDITION ←
 { RELOP NUMBER |
 call PROCEDURE_NAME }

WHERE-RESTRICTION ←
 where BOOLEAN

BOOLEAN ←
 { PREDICATE |
 (BOOLEAN) |
 not BOOLEAN |
 BOOLEAN and BOOLEAN |
 BOOLEAN or BOOLEAN }

PREDICATE ←
 { if CONDITION then PREDICATE [else PREDICATE] fi |
 CONDITION }

CONDITION ←
 { EXPRESSION RELOP EXPRESSION |
 present (EXPRESSION , STRING_LIST) |
 call PROCEDURE_NAME }

```
EXPRESSION  ←
      { SUBEXPRESSION |
        ( EXPRESSION ) |
        EXPRESSION RELOP EXPRESSION }

SUBEXPRESSION  ←
      { ATOMIC_EXPRESSION |
        SET_FUNCTION ( <EXPRESSION> ) |
        append ( EXPRESSION , EXPRESSION ) |
        substring ( EXPRESSION , CHAR_POS , CHAR_POS ) |
        left ( EXPRESSION , CHAR_POS ) |
        right ( EXPRESSION , CHAR_POS ) |
        location ( EXPRESSION , EXPRESSION ) |
        length ( EXPRESSION ) |
        repetitions LABEL through LABEL }

ATOMIC_EXPRESSION  ←
      { LABEL |
    ·   STRING |
        [{ + | - }] NUMBER |
        * }

CHAR_POS  ←
      { STRING |
        STRING { + | - } INTEGER |
        INTEGER }

ORDERING_CLAUSE  ←
      ordering : ORDERING

ORDERING  ←
      { <LABEL> |
        none |
        atomic |
        call PROCEDURE_NAME }

VIOLATION_ACTION_CLAUSE  ←
      { error [STRING] |
        call PROCEDURE_NAME }
```

Appendix 4: YACC SDML Grammar Specification

This Appendix contains the SDML grammar specification as given to the Yacc program.

```
sdm_schema :      class_definition_list
            ;

class_definition_list :    class_definition
            |         class_definition_list class_definition
            ;

class_definition :  base_class_definition
            |       nonbase_class_def
            ;

base_class_definition :    class_name_list base_class_body
            ;

class_name_list   :    CLASS_NAME
            |      class_name_list comma_and_sep CLASS_NAME
            ;

comma_and_sep     :    ','
            |      AND
            ;

base_class_body   :    desc_clause bc_feat_decl
                       member_attr_decl class_attr_decl ident_decl
            ;

desc_clause  :    /* empty */
            |     DESCRIPTION ':' DESCRIPTION_TEXT
            ;

bc_feat_decl :    /* empty */
            |     DUPLICATES ALLOWED
            |     DUPLICATES NOT ALLOWED
            ;

member_attr_decl : MEMBER_ATTRIBUTES ':' member_attr_list
            ;

member_attr_list : member_attribute
            |      member_attr_list member_attribute
            ;

member_attribute : attr_name_list desc_clause value_class_decl
```

```
                    inverse_decl match_or_derivation
                    member_order member_attr_opts
                    attr_assertion_decl
            ;

attr_name_list    :    ATTRIBUTE_NAME
            |    attr_name_list comma_and_sep ATTRIBUTE_NAME
            ;

value_class_decl :  VALUE_CLASS ':' CLASS_NAME
            ;

inverse_decl :    /* empty */
            |    INVERSE ':' ATTRIBUTE_NAME
            ;

match_or_derivation :    /* empty */
            |    match_decl
            |    derivation_decl
            ;

match_decl   :    MATCH ':' ATTRIBUTE_NAME OF CLASS_NAME ON ATTRIBUTE
            ;

derivation_decl    :    DERIVATION ':' member_attr_deriv
            ;

member_order    :    /* empty */
            |    SINGLE_VALUED
            |    MULTIVALUED
            |    MULTIVALUED WITH_SIZE_BETWEEN INTEGER_C AND INTEGR
            ;

member_attr_opts :/* empty */
            |    member_options
            ;

member_options    :    member_opt_item
            |    member_options member_opt_item
            ;

member_opt_item  :    MAY_NOT_BE_NULL
            |    NOT CHANGEABLE
            |    EXHAUSTS VALUE_CLASS
            |    NO_OVERLAP_IN_VALUES
            ;

attr_assertion_decl :    /* empty */
            |    attr_assertion_list fail_action_clause
            ;
```

```
attr_assertion_list : attribute_assertion
            |       attr_assertion_list attribute_assertion
            ;

attribute_assertion : ASSERTION ':' assertion
            ;

assertion       :       CALL PROCEDURE_NAME
            |       mapping_expression relop mapping_expression
            |       setop_predicate
            ;

fail_action_clause : /* empty */
            |       FAILURE_ACTION ':' failure_action
            ;

failure_action      :       CALL PROCEDURE_NAME
            |       ERROR action_message
            |       WARNING STRING_C
            ;

class_attr_decl     :       /* empty */
            |       CLASS_ATTRIBUTES ':' class_attr_list
            ;

class_attr_list     :       class_attribute
            |       class_attr_list class_attribute
            ;

class_attribute     :       attr_name_list desc_clause
                    value_class_decl class_deriv_decl
                    member_order class_attr_opts
                    attr_assertion_decl
            ;

class_attr_opts     :       /* empty */
            |       class_options
            ;

class_options:      class_opt_item
            |       class_options class_opt_item
            ;

class_opt_item      :       MAY_NOT_BE_NULL
            |       NOT CHANGEABLE
            ;

class_deriv_decl :  /* empty */
            |       DERIVATION ':' class_attr_deriv
            ;
```

```
member_attr_deriv :     interattr_deriv
            |       member_spec_deriv
            ;

member_spec_deriv :     ordering_predicate
            |       existence_predicate
            |       recursive_trace_pred
            ;

ordering_predicate : ORDER BY direction mapping_list within_clause
            ;

direction    :      /* empty */
             |      INCREASING
             |      DECREASING
             ;

within_clause       :      /* empty */
             |      WITHIN mapping_list
             ;

mapping_list :      mapping
             |      mapping_list comma_and_sep mapping
             ;

existence_predicate : IF IN CLASS_NAME
             ;

recursive_trace_pred :     level_clause LEVELS_OF_VALUES OF ATTRIBUTE_NAME
             ;

level_clause :      UP_TO INTEGER_C
             |      ALL
             ;

class_attr_deriv :  interattr_deriv
             |      class_spec_deriv
             ;

class_spec_deriv :  class_size_pred
             |      class_member_pred
             ;

class_size_pred     :       NUMBER OF uniqueness MEMBERS IN THIS_CLASS
             ;

uniqueness   :      /* empty */
             |      UNIQUE
             ;

class_member_pred :     set_function OF ATTRIBUTE_NAME OVER MEMBERS OF THIS
```

```
                ;

set_function :      MINIMUM
             |      MAXIMUM
             |      AVERAGE
             |      SUM
                ;

interattr_deriv    :      derived_predicate
             |      subvalue_predicate
             |      set_derived_pred
             |      equality_predicate
             |      set_order_predicate
                ;

derived_predicate :  SAME_AS mapping
                ;

set_derived_pred :  WHERE IS IN mapping AND IS IN mapping
             |      WHERE IS IN mapping OR IS IN mapping
             |      WHERE IS IN mapping AND IS NOT IN mapping
                ;

equality_predicate :  EQ mapping_expression
                ;

mapping_expression :      mapping_term
             |            mapping_expression addition_operator mapping_term
                ;

mapping_term    :      mapping_factor
             |         mapping_term multiply_operator mapping_factor
                ;

mapping_factor    :      mapping_primary
             |         mapping_factor exponent_operator mapping_primary
                ;

mapping_primary   :      '(' mapping_expression ')'
             |         set_function '(' mapping ')'
             |         mapping
             |         number
             |         CONST_MEMBER_NAME
             |         SIZEOF '(' mapping ')'
             |         SIZEOF '(' CLASS_NAME ')'
                ;

addition_operator :  '+'
                     { = '+'; }
             |       '-'
                     { = '-'; }
```

```
                ;
multiply_operator : '*'
                        { = '*'; }
             |      '/'
                        { = '/'; }
                ;

exponent_operator : '!'
                        { = '!'; }
                ;

set_order_predicate :       NUMBER OF uniqueness MEMBERS IN mapping
                ;

subvalue_predicate :SUBVALUE OF mapping WHERE subvalue_selection
                ;

subvalue_selection : IS IN CLASS_NAME
             |      attribute_predicate
                ;

mapping            :       ATTRIBUTE_NAME
             |      mapping '.' ATTRIBUTE_NAME
                ;

ident_decl    :      IDENTIFIERS ':' ident_list
             |       IDENTIFIERS ':' NONE
                ;

ident_list    :      identifier
             |       ident_list ',' identifier
                ;

identifier    :      ATTRIBUTE_NAME
             |       identifier '+' ATTRIBUTE_NAME
                ;

nonbase_class_def : class_name_list nonbase_class_body
                ;

nonbase_class_body :        desc_clause interclass_connection
                    nonbase_class_alts
                ;

nonbase_class_alts :nonbase_class_feat
             |      name_class_feat
                ;

nonbase_class_feat :        nbc_member_attr_decl class_attr_decl
                ;
```

- 134 -

```
nbc_member_attr_decl :   /* empty */
            |    member_attr_decl
            ;

name_class_feat :   determines_clause
            |    const_members_clause
            |    determines_clause const_members_clause
            ;

determines_clause : DETERMINES ':' CLASS_NAME
            ;

const_members_clause :   CONSTANT_MEMBERS ':' const_members_list
            ;

const_members_list :   CONST_MEMBER_NAME
            |    const_members_list comma_and_sep CONST_MEMBER_NAME
            ;

interclass_connection :   INTERCLASS_CONNECTION ':' connection
                    ic_assertion_decl
            ;

ic_assertion_decl :   /* empty */
            |    ic_assertion_list fail_action_clause
            ;

ic_assertion_list :   ic_assertion
            |    ic_assertion_list ic_assertion
            ;

ic_assertion  :       INTERCLASS_ASSERTION ':' connection
            ;

connection    :       subclass_connection
            |    grouping_connection
            ;

subclass_connection :   SUBCLASS OF CLASS_NAME subclass_predicate
            ;

grouping_connection :   expression_defined_gc
            |    enumerated_gc
            |    user_controllable_gc
            |    name_class_gc
            ;

subclass_predicate : WHERE attribute_predicate
            |    WHERE SPECIFIED
            |    WHERE set_oper_defined_sc
            |    WHERE existence_sc
```

- 135 -

```
                  |       name_class_sc
                  ;

set_oper_defined_sc :     IS IN CLASS_NAME AND IS IN CLASS_NAME
                  |       IS IN CLASS_NAME OR IS IN CLASS_NAME
                  |       IS NOT IN CLASS_NAME
                  ;

existence_sc :    IS A_VALUE OF ATTRIBUTE_NAME OF CLASS_NAME
                  ;

name_class_sc     :       /* empty */
                  |       WHERE FORMAT IS format_directive
                  ;

expression_defined_gc :   GROUPING OF CLASS_NAME ON_COMMON_VALUE OF
                          attr_name_list explicitly_def_grps
                  ;

explicitly_def_grps :     /* empty */
                  |       GROUPS_DEFINED_AS_CLASSES_ARE class_name_list
                  ;

enumerated_gc     :       GROUPING OF CLASS_NAME CONSISTING_OF_CLASSES
                          class_name_list
                  ;

user_controllable_gc :    GROUPING OF CLASS_NAME AS_SPECIFIED
                  ;

name_class_gc     :       GROUPING OF CLASS_NAME AS_SPECIFIED
                                  BY CLASS_NAME
                  ;

attribute_predicate : attr_pred_term
                  |       attribute_predicate OR attr_pred_term
                  ;

attr_pred_term    :       attr_pred_factor
                  |       attr_pred_term AND attr_pred_factor
                  ;

attr_pred_factor :  attr_pred_primary
                  |       NOT attr_pred_primary
                  ;

attr_pred_primary :       '(' attribute_predicate ')'
                  |       simple_predicate
                  ;

simple_predicate : relop_predicate
```

```
                    |       setop_predicate
                    ;

relop_predicate :   mapping relop constant
                |   mapping relop mapping
                ;

setop_predicate :   mapping setop constant
                |   mapping setop mapping
                |   mapping setop CLASS_NAME
                ;

relop           :   EQ
                |   NE
                |   LT
                |   LE
                |   GT
                |   GE
                ;

setop           :   IS properly CONTAINED IN
                |   properly CONTAINS
                ;

properly        :   /* empty */
                |   PROPERLY
                ;

format_directive :  description_clause ordering_clause
                    violation_actn_clause
                ;

description_clause : description_subclause
                |   description_clause OR description_subclause
                ;

description_subclause :   description '.'
                |   description ',' where_restriction '.'
                ;

description     :   subunit
                |   description ',' subunit
                ;

subunit             :       subunit_item
                |   LABEL ':' subunit_item
                ;

subunit_item:       STRING str_where_clause
                |   NUMBER num_where_clause
                |   ONEOF string_list
```

```
            |       ONEOF number_list
            |       STRING_C
            ;

str_where_clause : /* empty */
            |       WHERE string_boolean
            ;

num_where_clause :       /* empty */
            |       WHERE number_boolean
            ;

string_list  :       string_component
            |       '(' string_list ',' string_component ')'
            ;

string_component : STRING_C
            |       ALPHABETICS
            |       NUMERICS
            |       SPECIALS
            ;

number_list :       number
            |       '(' number_list ',' number ')'
            ;

string_boolean       :       string_bool_term
            |       string_boolean OR string_bool_term
            ;

string_bool_term :  string_bool_factor
            |       string_bool_term AND string_bool_factor
            ;

string_bool_factor : string_bool_primary
            |       NOT string_bool_primary
            ;

string_bool_primary :       '(' string_boolean ')'
            |       string_predicate
            ;

string_predicate :  IF string_condition THEN string_predicate FI
            |       IF string_condition THEN string_predicate
                    ELSE string_predicate FI
            |       string_condition
            ;

string_condition :  relop STRING_C
            |       SIZE relop INTEGER_C
            |       HAS string_list
```

```
               |     CALL PROCEDURE_NAME
               ;

number_boolean    :     number_bool_term
               |     number_boolean OR number_bool_term
               ;

number_bool_term :     number_bool_factor
               |     number_bool_term AND number_bool_factor
               ;

number_bool_factor :     number_bool_primary
               |     NOT number_bool_primary
               ;

number_bool_primary :   '(' number_boolean ')'
               |     number_predicate
               ;

number_predicate : IF number_condition THEN number_predicate FI
               |     IF number_condition THEN number_predicate
                      ELSE number_predicate FI
               |     INTEGER
               |     REAL
               |     number_condition
               ;

number_condition : relop number
               |     CALL PROCEDURE_NAME
               ;

where_restriction :  WHERE boolean
               ;

boolean           :     boolean_term
               |     boolean OR boolean_term
               ;

boolean_term      :     boolean_factor
               |     boolean_term AND boolean_factor
               ;

boolean_factor    :     boolean_primary
               |     NOT boolean_primary
               ;

boolean_primary   :     '(' boolean ')'
               |     predicate
               ;

predicate    :     IF condition_expr THEN predicate FI
```

- 139 -

```
               |        IF condition_expr THEN predicate ELSE predicate FI
               |        condition
               ;

condition      :        expression relop expression
               |        PRESENT '(' expression ',' expression ')'
               |        CALL PROCEDURE_NAME
               ;

condition_expr      :     condition_term
               |        condition_expr AND condition_term
               ;

condition_term      :     condition_factor
               |        condition_term OR condition_factor
               ;

condition_factor :      '(' condition_expr ')'
               |        condition
               ;

expression     :        arithmetic_term
               |        expression addition_operator arithmetic_term
               ;

arithmetic_term     :      arithmetic_factor
               |        arithmetic_term multiply_operator arithmetic_factor
               ;

arithmetic_factor :     arithmetic_primary
               |        arithmetic_factor exponent_operator arithmetic_primary
               ;

arithmetic_primary :       '(' expression ')'
               |        subexpression
               ;

subexpression:          atomic_expression
               |        set_function '(' expression_list ')'
               |        APPEND '(' expression ',' expression ')'
               |        SUBSTRING '(' expression ',' char_pos ','
                            char_pos ')'
               |        LEFT '(' expression ',' char_pos ')'
               |        RIGHT '(' expression ',' char_pos ')'
               |        LOCATION '(' expression ',' expression ')'
               |        LENGTH '(' expression ')'
               |        REPETITIONS LABEL THROUGH LABEL
               ;

atomic_expression : LABEL
               |        STRING_C
```

```
                    |      number
                    |      addition_operator number
                    |      '*'
                    ;

expression_list     :      expression
                    |      expression_list ',' expression
                    ;

char_pos            :      STRING_C
                    |      STRING_C addition_operator INTEGER_C
                    |      INTEGER_C
                    ;

ordering_clause     :      /* empty */
                    |      ORDERING ':' ordering
                    ;

ordering            :      ordering_list
                    |      NONE
                    |      ATOMIC
                    |      CALL PROCEDURE_NAME
                    ;

ordering_list       :      LABEL
                    |      ordering_list ',' LABEL
                    ;

violation_actn_clause :    /* empty */
                    |      VIOLATION_ACTION ':' violation_action
                    ;

violation_action    :      ERROR action_message
                    |      CALL PROCEDURE_NAME
                    ;

action_message      :      /* empty */
                    |      STRING_C
                    ;

number              :      INTEGER_C
                    |      REAL_C
                    ;

constant            :      STRING_C
                    |      number
                    |      CONST_MEMBER_NAME
                    ;
```

Appendix 5: Example SDML Specification

This Appendix contains the SDML specification for the SDM schema example in Appendix 1. The SDM schema in Appendix 1 was changed to adhere to the syntax of the SDML specification and was enhanced using some of the extensions provided in Chapter 3 of this paper.

```
CARS and AUTOMOBILES
   description: { All cars produced by a plant.
                 This is a base class definition. }
   duplicates not allowed
   member attributes:

      Make
         value class: CAR_TYPES
         may not be null
         not changeable

      Model
         value class: CAR_MODELS
         may not be null
         not changeable

      Year
         value class: YEARS
         may not be null
         not changeable

      Exterior_color
         value class: COLORS

      Interior_color
         value class: COLORS
         assertion: call _verify_interior_color
         failure action: error
             'invalid exterior/interior combination'

      Base_price
         value class: PRICES

      Sticker_price
         value class: PRICES
         derivation: = Base_price + sum( Options.Price )

      Options
         description: { Options on car }
         value class: OPTIONS
```

Dealership
    description: { Dealership stocking this car.
                  This attribute shows the inverse
                  relationship. Here, Dealership is
                  tbe DEALER with this car in stock. }
    value class: DEALERS
    inverse: Cars_in_stock

Vin
    description: { Vehicle Identification Number }
    value class: VIN_CODES
    may not be null
    not changeable

Owner
    description: { Ultimate owner of vehicle until purchased
                  by a consumer. This attribute shows
                  the matching relationship. Here,
                  Owner is the owner of the Dealership
                  which has this car in stock. }
    value class: OWNERS
    match: Owned_by of DEALERS on Cars_in_stock

identifiers:
    Vin

DEALERS, DEALERSHIPS
    description: { Car dealerships wbich stock the cars built by
              a plant. This is a base class definition. }
    member attributes:

Name
    description: { Dealership name }
    value class: DEALER_NAMES

Address
    value class: ADDRESS

Dealer_code
    description: { Code uniquely identifying a dealership }
    value class: DEALER_CODES
    may not be null

Makes
    description: { Makes of cars this dealer carries }
    value class: CAR_TYPES
    multivalued

Cars_in_stock
    description: { Cars which this dealership has in stock.
            This attribute closes the symmetrical
            inverse relationship with the Dealership
            attribute of class CARS. }
    value class: CARS
    inverse: Dealership
    multivalued
    assertion: Cars_in_stock.Make is contained in Makes

Owned_by
    description: { Owners of this dealership }
    value class: OWNERS
    inverse: Dealers_owned

Employees
    description: { Employees working as sales persons for
            this dealership. }
    value class: PERSON_NAMES
    multivalued with size between 1 and 20

class attributes:

Number_dealer_cars
    description: { Number of cars in dealership. }
    value class: INTEGERS
    derivation: number of members in Cars_in_stock

identifiers:
    Dealer_code

OWNERS
  description: { This class contains all owners of car
           dealerships. This is a base class definition. }
  member attributes:

Name
    value class: PERSON_NAMES

Address
    value class: ADDRESS

Dealers_owned
    description: { This attribute is the dealerships
            which this person or persons own. }
    value class: DEALERS
    inverse: Owned_by
    multivalued

identifiers:
    Name

OPTIONS
    description: { This base class contains all available
                car options. }
    member attributes:

        Type
            description: { Type of option }
            value class: OPTION_TYPES

        Price
            description: { Price of options }
            value class: PRICES

    identifiers:
        Type

CARS_SOLD
    description: { This class contains all cars sold to a
                customer. }
    interclass connection: subclass of CARS where specified
    interclass assertion: subclass of CARS where
                                is in PREPARED_CARS or
                                is in SCHEDULED_PREPS
    failure action: warning 'car not scheduled for preparation'
    member attributes:

        Sold_to
            description: { Customer buying the car. }
            value class: PERSON_NAMES
            may not be null

        Sold_by
            description: { Salesman selling car to customer. }
            value class: PERSON_NAMES
            may not be null
            assertion: Sold_by is contained in Dealership.Employees
            failure action:
                        error 'Salesman not employed by Dealership'

        Customer_address
            value class: ADDRESS

        Date_sold
            value class: DATES

        Selling_price
            description: { Final selling price of car. }
            value class: PRICES

class attributes:

Number_of_cars_sold
  value class: INTEGERS
  derivation: number of unique members in this class
  assertion: Number_of_cars_sold =
        sizeof(SCHEDULED_PREPS) +
        sizeof(PREPARED_CARS)
  failure action: call _determine_cars_not_scheduled

SCHEDULED_PREPS
  description: { This class contains all scheduled car
        preparations. This base class definition
        is an example of an event class definition. }
  interclass connection: subclass of CARS where specified
  member attributes:

Date_of_preparation
  value class: DATES
  assertion: Date_of_preparation <= _CURRENT_DATE
  failure action: call _notify_past_due

class attributes:

Number_of_sched_preps
  description: { Number of cars to be prepared }
  value class: INTEGERS
  derivation: number of unique members in this class

BUICKS
  description: { This class contains all Buick cars. This
        nonbase class shows the use of the
        attribute-defined subclass connection utilizing
        the simple attribute predicate. }
  interclass connection: subclass of CARS where Make = 'Buick'
  member attributes:

Model
  description: { This shows the concept of restricting
        the value class of an inherited member
        attribute. }
  value class: BUICK_MODELS

SOMERSETS
   description: { This class contains all Buick Somerset model
              cars. This nonbase class definition shows
              the use of the attribute-defined subclass
              connection utilizing the compound attribute
              predicate. }
   interclass connection: subclass of CARS where
                        Make = 'Buick' and
                        Model = 'Somerset'
   interclass assertion: subclass of BUICKS where
                        Model = 'Somerset'

PREPARED_CARS
   description: { This class contains all prepared cars for
              any dealership. This nonbase class definition
              shows the use of the user-controllable
              subclass connection. }
   interclass connection: subclass of CARS where specified

PREPARED_BUICKS
   description: { This class contains all prepared Buick's for
              any dealership. This nonbase class definition
              shows the use of the intersection
              set-operator-defined subclass connection. }
   interclass connection: subclass of CARS where
                      is in BUICKS and
                    is in PREPARED_CARS
   interclass assertion: subclass of PREPARED_CARS where
                        Make = 'Buick'

BUICK_DEALERS
   description: { This class contains all dealers which sell
              Buick's. This nonbase class definition shows
              the use of the existence subclass connection. }
   interclass connection: subclass of DEALERS where is a value of
                      Dealership of BUICKS
   interclass assertion: subclass of DEALERS where
                      Make contains 'Buick'

CAR_MODEL_GROUPS
   description: { This class groups cars into models. This
              nonbase class definition shows the use of the
              expression-defined grouping class. Each member
              of this nonbase class is a class containing all
              models for cars. Note that the class definition
              also indicates that a SDM class was explicitly
              defined for the Buick Somerset make and model. }
   interclass connection: grouping of BUICKS on common value of
                      Make and Model
  groups defined as classes are SOMERSETS

DEALER_PREP_CARS
   description: { This class groups prepared cars for specific
                dealerships. This nonbase class definition
                also shows the use of the expression-defined
                grouping class. }
   interclass connection: grouping of PREPARED_CARS on common value
                of Dealership

CAR_TYPES
   description: { This is the list of all available car types. }
   interclass connection: subclass of STRINGS where specified
   determines: CAR_MODELS

CAR_MODELS
   description: { This is the list of all available car models. }
   interclass connection: grouping of STRINGS where specified
                by CAR_TYPES

BUICK_MODELS
   description: { This is the list of all possible
            buick car models. }
   interclass connection: subclass of STRINGS where specified

YEARS
   description: { This is the format of a year. }
   interclass connection: subclass of STRINGS where format is
                number where integer and $>= 70$ and $<= 99$.
   constant members: _CURRENT_YEAR

DATES
   description: { Calendar dates in the range
             "1/1/70" to "12/31/99". }
   interclass connection: subclass of STRINGS where format is
                month: number where integer and
                    $>=1$ and $<=12$,
                '/',
                day: number where integer and
                  $>=1$ and $<=31$,
                '/',
                year: number where integer and
                  $>=70$ and $<=99$,
                where if (month=4 or month=5 or
                  month=9 or month=11) then
                    day $<=30$ fi and
                if month=2 then day $<=29$ fi.
                ordering: year, month, day
   constant members: _CURRENT_DATE

COLORS
   description: { This is the available interior and exterior
            car colors. }

- 148 -

interclass connection: subclass of STRINGS where specified

PRICES
  description: { This is the possible car prices. }
  interclass connection: subclass of STRINGS where format is
                        number where > = 0,
                        where length(right(*, '.'+1)) = 2
                              or not present(*, '.').
                        ordering: value

VIN_CODES
  description: { This is the format of the Vehicle Identification
              Number Codes. For example: 1G4NK27U1GC612345. }
  interclass connection: subclass of STRINGS where format is
                    '1G',
                    number where integer and > =0 and < =9,
                    string where has alphabetics and size = 2,
                    string where has numerics and size = 2,
           engine_code:oneof('U', 'G'),
                    number where integer and > =0 and < =9,
           model_year: string where has alphabetics and size = 1,
           assy_plant: string where has alphabetics and size = 1,
                    string where has numerics and size = 6.
           ordering: call _vin_ordering

DEALER_NAMES
  description: { Dealership names }
  interclass connection: subclass of STRINGS

ADDRESS
  description: { Street Address }
  interclass connection: subclass of STRINGS

DEALER_CODES
  description: { Dealership identifier codes }
  interclass connection: subclass of STRINGS where format is
                    number where integer.

PERSON_NAMES
  description: { Personal Names }
  interclass connection: subclass of STRINGS

OPTION_TYPES
  description: { Option types }
  interclass connection: subclass of STRINGS where specified

NAME

  sdml - Semantic Database Model (SDM) language compiler

SYNOPSIS

  sdml [-dvsl] [-o ofile] [file]

DESCRIPTION

  Sdml compiles the SDM specification from the given *file* (standard input
  default). The compilation involves two passes. Pass 1 will verify syntax of
  the SDM specification and pass 2 will verify semantics of the SDM
  specification. Types of errors reported are class or attribute names used but
  not defined, multiple class or attribute name definitions, and inconsistent
  usage of various SDM capabilities. An output specification may be generated
  from the SDML compiler by specifying the -o option and a *ofile* to write the
  file to.

  The available options are as follows:

  -d   generate a file named *sdm.ddl* which will contain the structures built
          during pass 2 of the sdml compiler. This file is used by the data
          dictionary generation program to build a static data dictionary from
          the SDM specification given. Note that this option does not make
          sense with the -1 option.

  -v   verbose mode. This option will result in a summary being directed to
          standard output with statistics on various label definitions and usage.

  -s   dumps the symbol table generated by pass 1 (and 2) of the sdml
          compiler to standard output. The symbol table will contain
          information about symbol definition and usage on a per symbol basis.

  -o   generates the compiled SDM specification and puts it into the file
          named *ofile*.

  -1   run pass 1 only. This option may be useful when attempting to
          resolve syntax errors in a SDM specification without going through
          pass 2 (verifying semantics) if error recovery is attempted by the
          SDML compiler.

  The SDML compiler pre-defines the maximum number of unique symbols,
  maximum number of class definitions, and maximum number of attribute
  definitions. The maximum number of unique symbol definitions (including
  constant symbols) can be increased from the default of 254 symbols (2^8-1)
  by using the

%h *hval*

control statement at the beginning of the SDM specification. This statement will increase the number of hash table entries possible from the default limit to the next highest power of 2 from *hval*. That is, the new limit L will be selected such that $2^{n-1}-1 \leqslant hval \leqslant 2^n - 1 = L$.

The maximum number of class definitions can be increased from the default of 50 by using the

%c *cval*

control statement, where *cval* is the new maximum number of class definitions. The maximum number of attribute definitions can be increased from the default of 100 by using the

%a *aval*

control statement, where *aval* is the new maximum number of attribute definitions.

SEE ALSO
R. V. Lane, *Semantic Database Model Language (SDML): Grammar Specification and Parser*

SEMANTIC DATABASE MODEL LANGUAGE (SDML):
GRAMMAR SPECIFICATION AND PARSER

by

RICHARD VERNON LANE

B. S., Michigan State University, 1980

AN ABSTRACT OF A MASTER'S THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computer Science

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1986

ABSTRACT

This paper describes the Semantic Database Model Language (SDML). SDML is a high-level language based on the Semantic Database Model (SDM). SDM will allow a database designer to model a database application while retaining application data meaning. Some extensions to the SDM are introduced and are incorporated into the SDML grammar. SDML is used as an intermediate in the automatic generation of a static data dictionary for the application environment. This paper discusses the design of a context-free grammar for SDML and the design of an LR(1) parser for SDML. The SDML parser will check syntactic and semantic correctness of an SDML specification.